Table of Contents

Introduction	1.1
About Yii	1.2
Upgrading from Version 1.1	1.3
Installing Yii	1.4
Running Applications	1.5
Saying Hello	1.6
Working with Forms	1.7
Working with Databases	1.8
Generating Code with Gii	1.9
Looking Ahead	1.10
Overview	1.11
Entry Scripts	1.12
Applications	1.13
Application Components	1.14
Controllers	1.15
Models	1.16
Views	1.17
Modules	1.18
Filters	1.19
Widgets	1.20
Assets	1.21
Extensions	1.22
Overview	1.23
Bootstrapping	1.24
Routing and URL Creation	1.25
Requests	1.26
Responses	1.27
Sessions and Cookies	1.28
Handling Errors	1.29
Logging	1.30

Components	1.31
Properties	1.32
Events	1.33
Behaviors	1.34
Configurations	1.35
Aliases	1.36
Class Autoloading	1.37
Service Locator	1.38
Dependency Injection Container	1.39
Data Access Objects	1.40
Query Builder	1.41
Active Record	1.42
Migrations	1.43
Sphinx	1.44
Redis	1.45
MongoDB	1.46
ElasticSearch	1.47
Creating Forms	1.48
Validating Input	1.49
Uploading Files	1.50
Collecting tabular input	1.51
Getting Data for Multiple Models	1.52
Data Formatting	1.53
Pagination	1.54
Sorting	1.55
Data Providers	1.56
Data Widgets	1.57
Working with Client Scripts	1.58
Theming	1.59
Authentication	1.60
Authorization	1.61
Working with Passwords	1.62
Auth Clients	1.63
Best Practices	1.64

Overview	1.65
Data Caching	1.66
Fragment Caching	1.67
Page Caching	1.68
HTTP Caching	1.69
Quick Start	1.70
Resources	1.71
Controllers	1.72
Routing	1.73
Response Formatting	1.74
Authentication	1.75
Rate Limiting	1.76
Versioning	1.77
Error Handling	1.78
Debug Toolbar and Debugger	1.79
Generating Code using Gii	1.80
Generating API Documentation	1.81
Overview	1.82
Testing environment setup	1.83
Unit Tests	1.84
Functional Tests	1.85
Acceptance Tests	1.86
Fixtures	1.87
Advanced Project Template	1.88
Building Application from Scratch	1.89
Console Commands	1.90
Core Validators	1.91
Internationalization	1.92
Mailing	1.93
Performance Tuning	1.94
Shared Hosting Environment	1.95
Template Engines	1.96
Working with Third-Party Code	1.97

GridView	1.98
ListView	1.99
DetailView	1.100
ActiveForm	1.101
Pjax	1.102
Menu	1.103
LinkPager	1.104
LinkSorter	1.105
Bootstrap Widgets	1.106
jQuery UI Widgets	1.107
Overview	1.108
ArrayHelper	1.109
Html	1.110
Url	1.111

The Definitive Guide to Yii 2.0

This tutorial is released under the Terms of Yii Documentation.

All Rights Reserved.

2014 (c) Yii Software LLC.

Introduction

- About Yii
- Upgrading from Version 1.1

Getting Started

- Installing Yii
- Running Applications
- Saying Hello
- Working with Forms
- Working with Databases
- Generating Code with Gii
- Looking Ahead

Application Structure

- Overview
- Entry Scripts
- Applications
- Application Components
- Controllers
- Models
- Views
- Modules
- Filters
- Widgets
- Assets
- Extensions

Handling Requests

- Overview
- Bootstrapping
- Routing and URL Creation
- Requests
- Responses
- Sessions and Cookies
- Handling Errors
- Logging

Key Concepts

- Components
- Properties
- Events
- Behaviors
- Configurations
- Aliases
- Class Autoloading
- Service Locator
- Dependency Injection Container

Working with Databases

- Data Access Objects: Connecting to a database, basic queries, transactions, and schema manipulation
- Query Builder: Querying the database using a simple abstraction layer
- Active Record: The Active Record ORM, retrieving and manipulating records, and defining relations
- Migrations: Apply version control to your databases in a team development environment
- Sphinx
- Redis
- MongoDB
- ElasticSearch

Getting Data from Users

- Creating Forms
- Validating Input
- Uploading Files
- Collecting Tabular Input
- Getting Data for Multiple Models

Displaying Data

- Data Formatting
- Pagination
- Sorting
- Data Providers
- Data Widgets
- Working with Client Scripts
- Theming

Security

- Overview
- Authentication
- Authorization
- Working with Passwords
- Cryptography
- Auth Clients
- Best Practices

Caching

- Overview
- Data Caching
- Fragment Caching
- Page Caching
- HTTP Caching

RESTful Web Services

- Quick Start

- Resources
- Controllers
- Routing
- Response Formatting
- Authentication
- Rate Limiting
- Versioning
- Error Handling

Development Tools

- Debug Toolbar and Debugger
- Generating Code using Gii
- Generating API Documentation

Testing

- Overview
- Testing environment setup
- Unit Tests
- Functional Tests
- Acceptance Tests
- Fixtures

Special Topics

- Advanced Project Template
- Building Application from Scratch
- Console Commands
- Core Validators
- Internationalization
- Mailing
- Performance Tuning
- Shared Hosting Environment
- Template Engines
- Working with Third-Party Code

Widgets

- GridView
- ListView
- DetailView
- ActiveForm
- Pjax
- Menu
- LinkPager
- LinkSorter
- Bootstrap Widgets
- jQuery UI Widgets

Helpers

- Overview
- ArrayHelper
- Html
- Url

What is Yii

Yii is a high performance, component-based PHP framework for rapidly developing modern Web applications. The name Yii (pronounced `Yee` or `[ji:]`) means "simple and evolutionary" in Chinese. It can also be thought of as an acronym for **Yes It Is**!

What is Yii Best for?

Yii is a generic Web programming framework, meaning that it can be used for developing all kinds of Web applications using PHP. Because of its component-based architecture and sophisticated caching support, it is especially suitable for developing large-scale applications such as portals, forums, content management systems (CMS), e-commerce projects, RESTful Web services, and so on.

How does Yii Compare with Other Frameworks?

If you're already familiar with another framework, you may appreciate knowing how Yii compares:

- Like most PHP frameworks, Yii implements the MVC (Model-View-Controller) architectural pattern and promotes code organization based on that pattern.
- Yii takes the philosophy that code should be written in a simple yet elegant way. Yii will never try to over-design things mainly for the purpose of strictly following some design pattern.
- Yii is a full-stack framework providing many proven and ready-to-use features: query builders and ActiveRecord for both relational and NoSQL databases; RESTful API development support; multi-tier caching support; and more.
- Yii is extremely extensible. You can customize or replace nearly every piece of the core's code. You can also take advantage of Yii's solid extension architecture to use or develop redistributable extensions.
- High performance is always a primary goal of Yii.

Yii is not a one-man show, it is backed up by a strong core developer team, as well as a large community of professionals constantly contributing to Yii's development. The Yii developer team keeps a close eye on the latest Web development trends and on the best

practices and features found in other frameworks and projects. The most relevant best practices and features found elsewhere are regularly incorporated into the core framework and exposed via simple and elegant interfaces.

Yii Versions

Yii currently has two major versions available: 1.1 and 2.0. Version 1.1 is the old generation and is now in maintenance mode. Version 2.0 is a complete rewrite of Yii, adopting the latest technologies and protocols, including Composer, PSR, namespaces, traits, and so forth. Version 2.0 represents the current generation of the framework and will receive the main development efforts over the next few years. This guide is mainly about version 2.0.

Requirements and Prerequisites

Yii 2.0 requires PHP 5.4.0 or above and runs best with the latest version of PHP 7. You can find more detailed requirements for individual features by running the requirement checker included in every Yii release.

Using Yii requires basic knowledge of object-oriented programming (OOP), as Yii is a pure OOP-based framework. Yii 2.0 also makes use of the latest features of PHP, such as namespaces and traits. Understanding these concepts will help you more easily pick up Yii 2.0.

Upgrading from Version 1.1

There are many differences between versions 1.1 and 2.0 of Yii as the framework was completely rewritten for 2.0. As a result, upgrading from version 1.1 is not as trivial as upgrading between minor versions. In this guide you'll find the major differences between the two versions.

If you have not used Yii 1.1 before, you can safely skip this section and turn directly to "Getting started".

Please note that Yii 2.0 introduces more new features than are covered in this summary. It is highly recommended that you read through the whole definitive guide to learn about them all. Chances are that some features you previously had to develop for yourself are now part of the core code.

Installation

Yii 2.0 fully embraces Composer, the de facto PHP package manager. Installation of the core framework, as well as extensions, are handled through Composer. Please refer to the Installing Yii section to learn how to install Yii 2.0. If you want to create new extensions, or turn your existing 1.1 extensions into 2.0-compatible extensions, please refer to the Creating Extensions section of the guide.

PHP Requirements

Yii 2.0 requires PHP 5.4 or above, which is a huge improvement over PHP version 5.2 that is required by Yii 1.1. As a result, there are many differences on the language level that you should pay attention to. Below is a summary of the major changes regarding PHP:

- Namespaces.
- Anonymous functions.
- Short array syntax `[...elements...]` is used instead of `array(...elements...)`.
- Short echo tags `<?=` are used in view files. This is safe to use starting from PHP 5.4.
- SPL classes and interfaces.
- Late Static Bindings.
- Date and Time.
- Traits.
- intl. Yii 2.0 makes use of the `intl` PHP extension to support internationalization

features.

Namespace

The most obvious change in Yii 2.0 is the use of namespaces. Almost every core class is namespaced, e.g., `yii\web\Request`. The "C" prefix is no longer used in class names. The naming scheme now follows the directory structure. For example, `yii\web\Request` indicates that the corresponding class file is `web/Request.php` under the Yii framework folder.

(You can use any core class without explicitly including that class file, thanks to the Yii class loader.)

Component and Object

Yii 2.0 breaks the `CComponent` class in 1.1 into two classes: [[yii\base\Object]] and [[yii\base\Component]]. The [[yii\base\Object|Object]] class is a lightweight base class that allows defining object properties via getters and setters. The [[yii\base\Component|Component]] class extends from [[yii\base\Object|Object]] and supports events and behaviors.

If your class does not need the event or behavior feature, you should consider using [[yii\base\Object|Object]] as the base class. This is usually the case for classes that represent basic data structures.

Object Configuration

The [[yii\base\Object|Object]] class introduces a uniform way of configuring objects. Any descendant class of [[yii\base\Object|Object]] should declare its constructor (if needed) in the following way so that it can be properly configured:

```
class MyClass extends \yii\base\Object
{
    public function __construct($param1, $param2, $config = [])
    {
        // ... initialization before configuration is applied

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();

        // ... initialization after configuration is applied
    }
}
```

In the above, the last parameter of the constructor must take a configuration array that contains name-value pairs for initializing the properties at the end of the constructor. You can override the [[yii\base\Object::init()|init()]] method to do initialization work that should be done after the configuration has been applied.

By following this convention, you will be able to create and configure new objects using a configuration array:

```
$object = Yii::createObject([
    'class' => 'MyClass',
    'property1' => 'abc',
    'property2' => 'cde',
], [$param1, $param2]);
```

More details about configurations can be found in the Configurations section.

Events

In Yii 1, events were created by defining an `on`-method (e.g., `onBeforeSave`). In Yii 2, you can now use any event name. You trigger an event by calling the [[yii\base\Component::trigger()|trigger()]] method:

```
$event = new \yii\base\Event;
$component->trigger($eventName, $event);
```

To attach a handler to an event, use the [[yii\base\Component::on()|on()]] method:

```
$component->on($eventName, $handler);
// To detach the handler, use:
// $component->off($eventName, $handler);
```

There are many enhancements to the event features. For more details, please refer to the Events section.

Path Aliases

Yii 2.0 expands the usage of path aliases to both file/directory paths and URLs. Yii 2.0 also now requires an alias name to start with the `@` character, to differentiate aliases from normal file/directory paths or URLs. For example, the alias `@yii` refers to the Yii installation directory. Path aliases are supported in most places in the Yii core code. For example, [[yii\caching\FileCache::cachePath]] can take both a path alias and a normal directory path.

A path alias is also closely related to a class namespace. It is recommended that a path alias be defined for each root namespace, thereby allowing you to use Yii class autoloader without any further configuration. For example, because `@yii` refers to the Yii installation directory, a class like `yii\web\Request` can be autoloaded. If you use a third party library, such as the Zend Framework, you may define a path alias `@Zend` that refers to that framework's installation directory. Once you've done that, Yii will be able to autoload any class in that Zend Framework library, too.

More on path aliases can be found in the Aliases section.

Views

The most significant change about views in Yii 2 is that the special variable `$this` in a view no longer refers to the current controller or widget. Instead, `$this` now refers to a *view* object, a new concept introduced in 2.0. The *view* object is of type [[yii\web\View]], which represents the view part of the MVC pattern. If you want to access the controller or widget in a view, you can use `$this->context`.

To render a partial view within another view, you use `$this->render()`, not `$this->renderPartial()`. The call to `render` also now has to be explicitly echoed, as the `render()` method returns the rendering result, rather than directly displaying it. For example:

```
echo $this->render('_item', ['item' => $item]);
```

Besides using PHP as the primary template language, Yii 2.0 is also equipped with official support for two popular template engines: Smarty and Twig. The Prado template engine is no longer supported. To use these template engines, you need to configure the `view` application component by setting the [[yii\base\View::$renderers|View::$renderers]] property. Please refer to the Template Engines section for more details.

Models

Yii 2.0 uses [[yii\base\Model]] as the base model, similar to `CModel` in 1.1. The class `CFormModel` has been dropped entirely. Instead, in Yii 2 you should extend [[yii\base\Model]] to create a form model class.

Yii 2.0 introduces a new method called [[yii\base\Model::scenarios()|scenarios()]] to declare supported scenarios, and to indicate under which scenario an attribute needs to be validated, can be considered as safe or not, etc. For example:

```
public function scenarios()
{
    return [
        'backend' => ['email', 'role'],
        'frontend' => ['email', '!role'],
    ];
}
```

In the above, two scenarios are declared: `backend` and `frontend`. For the `backend` scenario, both the `email` and `role` attributes are safe, and can be massively assigned. For the `frontend` scenario, `email` can be massively assigned while `role` cannot. Both `email` and `role` should be validated using rules.

The [[yii\base\Model::rules()|rules()]] method is still used to declare the validation rules. Note that due to the introduction of [[yii\base\Model::scenarios()|scenarios()]], there is no longer an `unsafe` validator.

In most cases, you do not need to override [[yii\base\Model::scenarios()|scenarios()]] if the [[yii\base\Model::rules()|rules()]] method fully specifies the scenarios that will exist, and if there is no need to declare `unsafe` attributes.

To learn more details about models, please refer to the Models section.

Controllers

Yii 2.0 uses [[yii\web\Controller]] as the base controller class, which is similar to `CController` in Yii 1.1. [[yii\base\Action]] is the base class for action classes.

The most obvious impact of these changes on your code is that a controller action should return the content that you want to render instead of echoing it:

```
public function actionView($id)
{
    $model = \app\models\Post::findOne($id);
    if ($model) {
        return $this->render('view', ['model' => $model]);
    } else {
        throw new \yii\web\NotFoundHttpException;
    }
}
```

Please refer to the Controllers section for more details about controllers.

Widgets

Yii 2.0 uses [[yii\base\Widget]] as the base widget class, similar to `CWidget` in Yii 1.1.

To get better support for the framework in IDEs, Yii 2.0 introduces a new syntax for using widgets. The static methods [[yii\base\Widget::begin()|begin()]], [[yii\base\Widget::end()|end()]], and [[yii\base\Widget::widget()|widget()]] have been introduced, to be used like so:

```
use yii\widgets\Menu;
use yii\widgets\ActiveForm;

// Note that you have to "echo" the result to display it
echo Menu::widget(['items' => $items]);

// Passing an array to initialize the object properties
$form = ActiveForm::begin([
    'options' => ['class' => 'form-horizontal'],
    'fieldConfig' => ['inputOptions' => ['class' => 'input-xlarge']],
]);
... form input fields here ...
ActiveForm::end();
```

Please refer to the Widgets section for more details.

Themes

Themes work completely differently in 2.0. They are now based on a path mapping mechanism that maps a source view file path to a themed view file path. For example, if the path map for a theme is `['/web/views' => '/web/themes/basic']`, then the themed version for the view file `/web/views/site/index.php` will be `/web/themes/basic/site/index.php`. For this reason, themes can now be applied to any view file, even a view rendered outside of the context of a controller or a widget.

Also, there is no more `CThemeManager` component. Instead, `theme` is a configurable property of the `view` application component.

Please refer to the Theming section for more details.

Console Applications

Console applications are now organized as controllers, like Web applications. Console controllers should extend from [[yii\console\Controller]], similar to `CConsoleCommand` in 1.1.

To run a console command, use `yii <route>`, where `<route>` stands for a controller route (e.g. `sitemap/index`). Additional anonymous arguments are passed as the parameters to the corresponding controller action method, while named arguments are parsed according to the declarations in [[yii\console\Controller::options()]].

Yii 2.0 supports automatic generation of command help information from comment blocks.

Please refer to the Console Commands section for more details.

I18N

Yii 2.0 removes the built-in date formatter and number formatter pieces in favor of the PECL intl PHP module.

Message translation is now performed via the `i18n` application component. This component manages a set of message sources, which allows you to use different message sources based on message categories.

Please refer to the Internationalization section for more details.

Action Filters

Action filters are implemented via behaviors now. To define a new, custom filter, extend from [[yii\base\ActionFilter]]. To use a filter, attach the filter class to the controller as a behavior. For example, to use the [[yii\filters\AccessControl]] filter, you would have the following code in a controller:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => 'yii\filters\AccessControl',
            'rules' => [
                ['allow' => true, 'actions' => ['admin'], 'roles' => ['@']],
            ],
        ],
    ];
}
```

Please refer to the Filtering section for more details.

Assets

Yii 2.0 introduces a new concept called *asset bundle* that replaces the script package concept found in Yii 1.1.

An asset bundle is a collection of asset files (e.g. JavaScript files, CSS files, image files, etc.) within a directory. Each asset bundle is represented as a class extending [[yii\web\AssetBundle]]. By registering an asset bundle via [[yii\web\AssetBundle::register()]], you make the assets in that bundle accessible via the Web. Unlike in Yii 1, the page registering the bundle will automatically contain the references to the JavaScript and CSS files specified in that bundle.

Please refer to the Managing Assets section for more details.

Helpers

Yii 2.0 introduces many commonly used static helper classes, including.

- [[yii\helpers\Html]]
- [[yii\helpers\ArrayHelper]]
- [[yii\helpers\StringHelper]]
- [[yii\helpers\FileHelper]]
- [[yii\helpers\Json]]

Please refer to the Helper Overview section for more details.

Forms

Yii 2.0 introduces the *field* concept for building a form using [[yii\widgets\ActiveForm]]. A field is a container consisting of a label, an input, an error message, and/or a hint text. A field is represented as an [[yii\widgets\ActiveField|ActiveField]] object. Using fields, you can build a form more cleanly than before:

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <div class="form-group">
        <?= Html::submitButton('Login') ?>
    </div>
<?php yii\widgets\ActiveForm::end(); ?>
```

Please refer to the Creating Forms section for more details.

Query Builder

In 1.1, query building was scattered among several classes, including `CDbCommand`, `CDbCriteria`, and `CDbCommandBuilder`. Yii 2.0 represents a DB query in terms of a [[yii\db\Query|Query]] object that can be turned into a SQL statement with the help of [[yii\db\QueryBuilder|QueryBuilder]] behind the scene. For example:

```
$query = new \yii\db\Query();
$query->select('id, name')
      ->from('user')
      ->limit(10);

$command = $query->createCommand();
$sql = $command->sql;
$rows = $command->queryAll();
```

Best of all, such query building methods can also be used when working with Active Record.

Please refer to the Query Builder section for more details.

Active Record

Yii 2.0 introduces a lot of changes to Active Record. The two most obvious ones involve query building and relational query handling.

The `CDbCriteria` class in 1.1 is replaced by [[yii\db\ActiveQuery]] in Yii 2. That class extends from [[yii\db\Query]], and thus inherits all query building methods. You call [[yii\db\ActiveRecord::find()]] to start building a query:

```
// To retrieve all *active* customers and order them by their ID:
$customers = Customer::find()
    ->where(['status' => $active])
    ->orderBy('id')
    ->all();
```

To declare a relation, simply define a getter method that returns an [[yii\db\ActiveQuery|ActiveQuery]] object. The property name defined by the getter represents the relation name. For example, the following code declares an `orders` relation (in 1.1, you would have to declare relations in a central place `relations()`):

```
class Customer extends \yii\db\ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany('Order', ['customer_id' => 'id']);
    }
}
```

Now you can use `$customer->orders` to access a customer's orders from the related table. You can also use the following code to perform an on-the-fly relational query with a customized query condition:

```
$orders = $customer->getOrders()->andWhere('status=1')->all();
```

When eager loading a relation, Yii 2.0 does it differently from 1.1. In particular, in 1.1 a JOIN query would be created to select both the primary and the relational records. In Yii 2.0, two SQL statements are executed without using JOIN: the first statement brings back the primary records and the second brings back the relational records by filtering with the primary keys of the primary records.

Instead of returning [[yii\db\ActiveRecord|ActiveRecord]] objects, you may chain the [[yii\db\ActiveQuery::asArray()|asArray()]] method when building a query to return a large number of records. This will cause the query result to be returned as arrays, which can significantly reduce the needed CPU time and memory if large number of records . For example:

```
$customers = Customer::find()->asArray()->all();
```

Another change is that you can't define attribute default values through public properties anymore. If you need those, you should set them in the init method of your record class.

```
public function init()
{
    parent::init();
    $this->status = self::STATUS_NEW;
}
```

There were some problems with overriding the constructor of an ActiveRecord class in 1.1. These are not present in version 2.0 anymore. Note that when adding parameters to the constructor you might have to override [[yii\db\ActiveRecord::instantiate()]].

There are many other changes and enhancements to Active Record. Please refer to the Active Record section for more details.

Active Record Behaviors

In 2.0, we have dropped the base behavior class `CActiveRecordBehavior`. If you want to create an Active Record Behavior, you will have to extend directly from `yii\base\Behavior`. If the behavior class needs to respond to some events of the owner, you have to override the `events()` method like the following:

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

User and IdentityInterface

The `CWebUser` class in 1.1 is now replaced by [[yii\web\User]], and there is no more `CUserIdentity` class. Instead, you should implement the [[yii\web\IdentityInterface]] which is much more straightforward to use. The advanced project template provides such an example.

Please refer to the Authentication, Authorization, and Advanced Project Template sections for more details.

URL Management

URL management in Yii 2 is similar to that in 1.1. A major enhancement is that URL management now supports optional parameters. For example, if you have a rule declared as follows, then it will match both `post/popular` and `post/1/popular`. In 1.1, you would have had to use two rules to achieve the same goal.

```
[
    'pattern' => 'post/<page:\d+>/<tag>',
    'route' => 'post/index',
    'defaults' => ['page' => 1],
]
```

Please refer to the Url manager docs section for more details.

An important change in the naming convention for routes is that camel case names of controllers and actions are now converted to lower case where each word is separated by a hypen, e.g. the controller id for the `CamelCaseController` will be `camel-case`. See the section about controller IDs and action IDs for more details.

Using Yii 1.1 and 2.x together

If you have legacy Yii 1.1 code that you want to use together with Yii 2.0, please refer to the Using Yii 1.1 and 2.0 Together section.

Installing Yii

You can install Yii in two ways, using the Composer package manager or by downloading an archive file. The former is the preferred way, as it allows you to install new extensions or update Yii by simply running a single command.

Standard installations of Yii result in both the framework and a project template being downloaded and installed. A project template is a working Yii project implementing some basic features, such as login, contact form, etc. Its code is organized in a recommended way. Therefore, it can serve as a good starting point for your projects.

In this and the next few sections, we will describe how to install Yii with the so-called *Basic Project Template* and how to implement new features on top of this template. Yii also provides another template called the Advanced Project Template which is better used in a team development environment to develop applications with multiple tiers.

> Info: The Basic Project Template is suitable for developing 90 percent of Web applications. It differs from the Advanced Project Template mainly in how their code is organized. If you are new to Yii, we strongly recommend you stick to the Basic Project Template for its simplicity yet sufficient functionalities.

Installing via Composer

Installing Composer

If you do not already have Composer installed, you may do so by following the instructions at getcomposer.org. On Linux and Mac OS X, you'll run the following commands:

```
curl -sS https://getcomposer.org/installer | php
mv composer.phar /usr/local/bin/composer
```

On Windows, you'll download and run Composer-Setup.exe.

Please refer to the Troubleshooting section of the Composer Documentation if you encounter any problems. If you are new to Composer, we also recommend to read at least the Basic usage section of the Composer documentation.

In this guide all composer commands assume you have installed composer globally so that it is available as the `composer` command. If you are using the `composer.phar` in the local directory instead, you have to adjust the example commands accordingly.

If you had Composer already installed before, make sure you use an up to date version. You can update Composer by running `composer self-update`.

> Note: During the installation of Yii, Composer will need to request a lot of information from the Github API. The number of requests depends on the number of dependencies your application has and may be bigger than the **Github API rate limit**. If you hit this limit, Composer may ask for your Github login credentials to obtain a Github API access token. On fast connections you may hit this limit earlier than Composer can handle so we recommend to configure the access token before installing Yii. Please refer to the Composer documentation about Github API tokens for instructions on how to do this.

Installing Yii

With Composer installed, you can install Yii by running the following commands under a Web-accessible folder:

```
composer global require "fxp/composer-asset-plugin:^1.2.0"
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

The first command installs the composer asset plugin which allows managing bower and npm package dependencies through Composer. You only need to run this command once for all. The second command installs the latest stable version of Yii in a directory named `basic`. You can choose a different directory name if you want.

> Info: If the `composer create-project` command fails make sure you have the composer asset plugin installed correctly. You can do that by running `composer global show`, which should contain an entry `fxp/composer-asset-plugin`. You may also refer to the Troubleshooting section of the Composer Documentation for common errors. When you have fixed the error, you can resume the aborted installation by running `composer update` inside of the `basic` directory.
>
> Tip: If you want to install the latest development version of Yii, you may use the following command instead, which adds a stability option:
>
> ```
> composer create-project --prefer-dist --stability=dev yiisoft/yii2-app-basic basi
> c
> ```
>
> Note that the development version of Yii should not be used for production as it may break your running code.

Installing from an Archive File

Installing Yii from an archive file involves three steps:

1. Download the archive file from yiiframework.com.
2. Unpack the downloaded file to a Web-accessible folder.
3. Modify the `config/web.php` file by entering a secret key for the `cookieValidationKey` configuration item (this is done automatically if you are installing Yii using Composer):

```
// !!! insert a secret key in the following (if it is empty) - this is required by
 cookie validation
'cookieValidationKey' => 'enter your secret key here',
```

Other Installation Options

The above installation instructions show how to install Yii, which also creates a basic Web application that works out of the box. This approach is a good starting point for most projects, either small or big. It is especially suitable if you just start learning Yii.

But there are other installation options available:

- If you only want to install the core framework and would like to build an entire application from scratch, you may follow the instructions as explained in Building Application from Scratch.
- If you want to start with a more sophisticated application, better suited to team development environments, you may consider installing the Advanced Project Template.

Verifying the Installation

After installation is done, either configure your web server (see next section) or use the built-in PHP web server by running the following console command while in the project `web` directory:

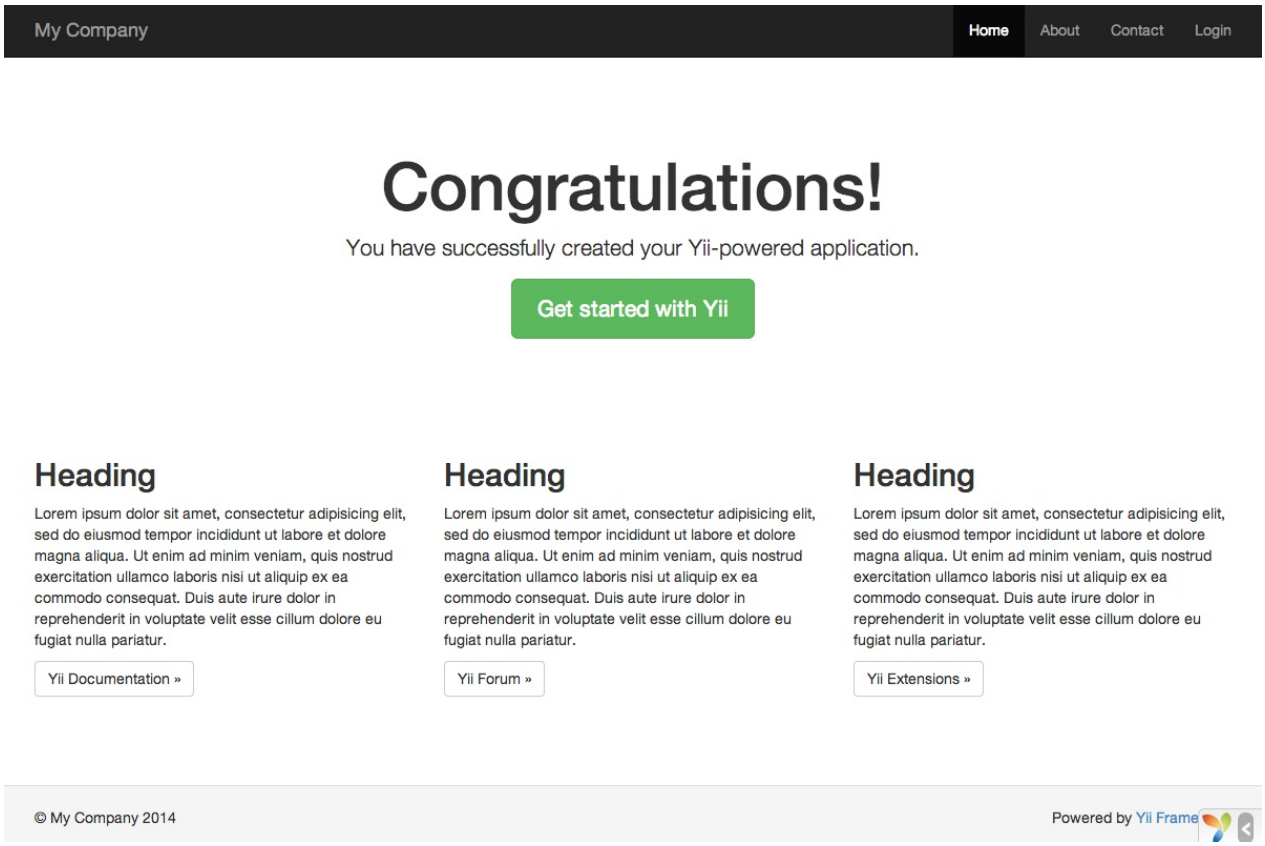
```
php yii serve
```

> Note: By default the HTTP-server will listen to port 8080. However if that port is already in use or you wish to serve multiple applications this way, you might want to specify what port to use. Just add the --port argument:

```
php yii serve --port=8888
```

You can use your browser to access the installed Yii application with the following URL:

```
http://localhost:8080/
```



You should see the above "Congratulations!" page in your browser. If not, please check if your PHP installation satisfies Yii's requirements. You can check if the minimum requirements are met using one of the following approaches:

- Copy `/requirements.php` to `/web/requirements.php` and then use a browser to access it via `http://localhost/requirements.php`
- Run the following commands:

```
cd basic
php requirements.php
```

You should configure your PHP installation so that it meets the minimum requirements of Yii. Most importantly, you should have PHP 5.4 or above. Ideally latest PHP 7. You should also install the PDO PHP Extension and a corresponding database driver (such as `pdo_mysql` for MySQL databases), if your application needs a database.

Configuring Web Servers

> Info: You may skip this subsection for now if you are just test driving Yii with no intention of deploying it to a production server.

The application installed according to the above instructions should work out of box with either an Apache HTTP server or an Nginx HTTP server, on Windows, Mac OS X, or Linux running PHP 5.4 or higher. Yii 2.0 is also compatible with facebook's HHVM. However, there are some edge cases where HHVM behaves different than native PHP, so you have to take some extra care when using HHVM.

On a production server, you may want to configure your Web server so that the application can be accessed via the URL `http://www.example.com/index.php` instead of `http://www.example.com/basic/web/index.php`. Such configuration requires pointing the document root of your Web server to the `basic/web` folder. You may also want to hide `index.php` from the URL, as described in the Routing and URL Creation section. In this subsection, you'll learn how to configure your Apache or Nginx server to achieve these goals.

> Info: By setting `basic/web` as the document root, you also prevent end users from accessing your private application code and sensitive data files that are stored in the sibling directories of `basic/web`. Denying access to those other folders is a security improvement.
>
> Info: If your application will run in a shared hosting environment where you do not have permission to modify its Web server configuration, you may still adjust the structure of your application for better security. Please refer to the Shared Hosting Environment section for more details.

Recommended Apache Configuration

Use the following configuration in Apache's `httpd.conf` file or within a virtual host configuration. Note that you should replace `path/to/basic/web` with the actual path for `basic/web`.

```
# Set document root to be "basic/web"
DocumentRoot "path/to/basic/web"

<Directory "path/to/basic/web">
    # use mod_rewrite for pretty URL support
    RewriteEngine on
    # If a directory or a file exists, use the request directly
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    # Otherwise forward the request to index.php
    RewriteRule . index.php

    # ...other settings...
</Directory>
```

Recommended Nginx Configuration

To use Nginx, you should install PHP as an FPM SAPI. You may use the following Nginx configuration, replacing `path/to/basic/web` with the actual path for `basic/web` and `mysite.local` with the actual hostname to serve.

```
server {
    charset utf-8;
    client_max_body_size 128M;

    listen 80; ## listen for ipv4
    #listen [::]:80 default_server ipv6only=on; ## listen for ipv6

    server_name mysite.local;
    root        /path/to/basic/web;
    index       index.php;

    access_log  /path/to/basic/log/access.log;
    error_log   /path/to/basic/log/error.log;

    location / {
        # Redirect everything that isn't a real file to index.php
        try_files $uri $uri/ /index.php$is_args$args;
    }

    # uncomment to avoid processing of calls to non-existing static files by Yii
    #location ~ \.(js|css|png|jpg|gif|swf|ico|pdf|mov|fla|zip|rar)$ {
    #    try_files $uri =404;
    #}
    #error_page 404 /404.html;

    # deny accessing php files for the /assets directory
    location ~ ^/assets/.*\.php$ {
        deny all;
    }

    location ~ \.php$ {
        include fastcgi_params;
        fastcgi_param SCRIPT_FILENAME $document_root$fastcgi_script_name;
        fastcgi_pass 127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
        try_files $uri =404;
    }

    location ~* /\. {
        deny all;
    }
}
```

When using this configuration, you should also set `cgi.fix_pathinfo=0` in the `php.ini` file in order to avoid many unnecessary system `stat()` calls.

Also note that when running an HTTPS server, you need to add `fastcgi_param HTTPS on;` so that Yii can properly detect if a connection is secure.

Running Applications

After installing Yii, you have a working Yii application that can be accessed via the URL `http://hostname/basic/web/index.php` or `http://hostname/index.php`, depending upon your configuration. This section will introduce the application's built-in functionality, how the code is organized, and how the application handles requests in general.

> Info: For simplicity, throughout this "Getting Started" tutorial, it's assumed that you have set `basic/web` as the document root of your Web server, and configured the URL for accessing your application to be `http://hostname/index.php` or something similar. For your needs, please adjust the URLs in our descriptions accordingly.

Note that unlike framework itself, after project template is installed it's all yours. You're free to add or delete code and overall modify it as you need.

Functionality

The basic application installed contains four pages:

- the homepage, displayed when you access the URL `http://hostname/index.php`,
- the "About" page,
- the "Contact" page, which displays a contact form that allows end users to contact you via email,
- and the "Login" page, which displays a login form that can be used to authenticate end users. Try logging in with "admin/admin", and you will find the "Login" main menu item will change to "Logout".

These pages share a common header and footer. The header contains a main menu bar to allow navigation among different pages.

You should also see a toolbar at the bottom of the browser window. This is a useful debugger tool provided by Yii to record and display a lot of debugging information, such as log messages, response statuses, the database queries run, and so on.

Additionally to the web application, there is a console script called `yii`, which is located in the applications base directory. This script can be used to run background and maintenance tasks for the application, which are described in the Console Application Section.

Application Structure

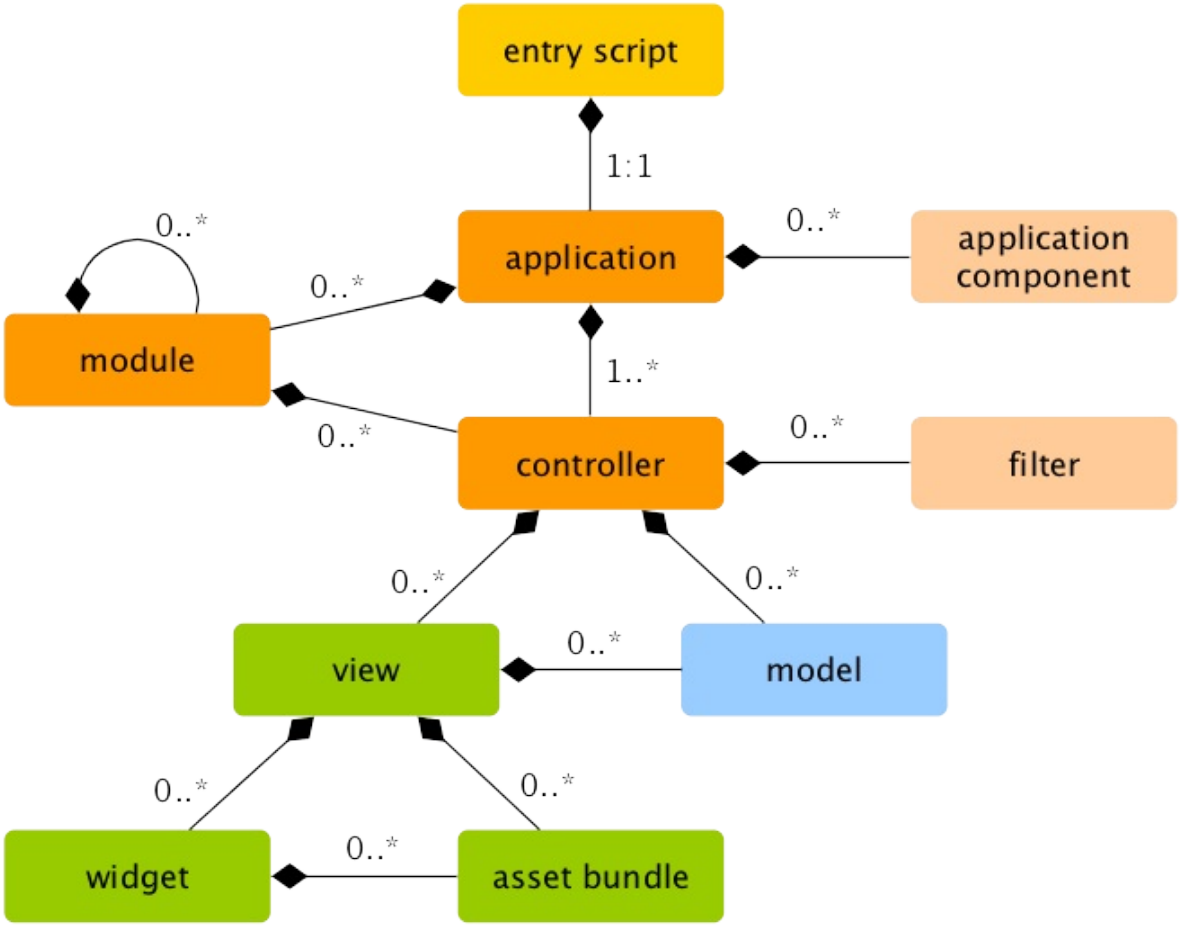
The most important directories and files in your application are (assuming the application's root directory is `basic`):

```
basic/                  application base path
    composer.json       used by Composer, describes package information
    config/             contains application and other configurations
        console.php     the console application configuration
        web.php         the Web application configuration
    commands/           contains console command classes
    controllers/        contains controller classes
    models/             contains model classes
    runtime/            contains files generated by Yii during runtime, such as logs a
nd cache files
    vendor/             contains the installed Composer packages, including the Yii fr
amework itself
    views/              contains view files
    web/                application Web root, contains Web accessible files
        assets/         contains published asset files (javascript and css) by Yii
        index.php       the entry (or bootstrap) script for the application
    yii                 the Yii console command execution script
```

In general, the files in the application can be divided into two types: those under `basic/web` and those under other directories. The former can be directly accessed via HTTP (i.e., in a browser), while the latter can not and should not be.

Yii implements the model-view-controller (MVC) architectural pattern, which is reflected in the above directory organization. The `models` directory contains all model classes, the `views` directory contains all view scripts, and the `controllers` directory contains all controller classes.

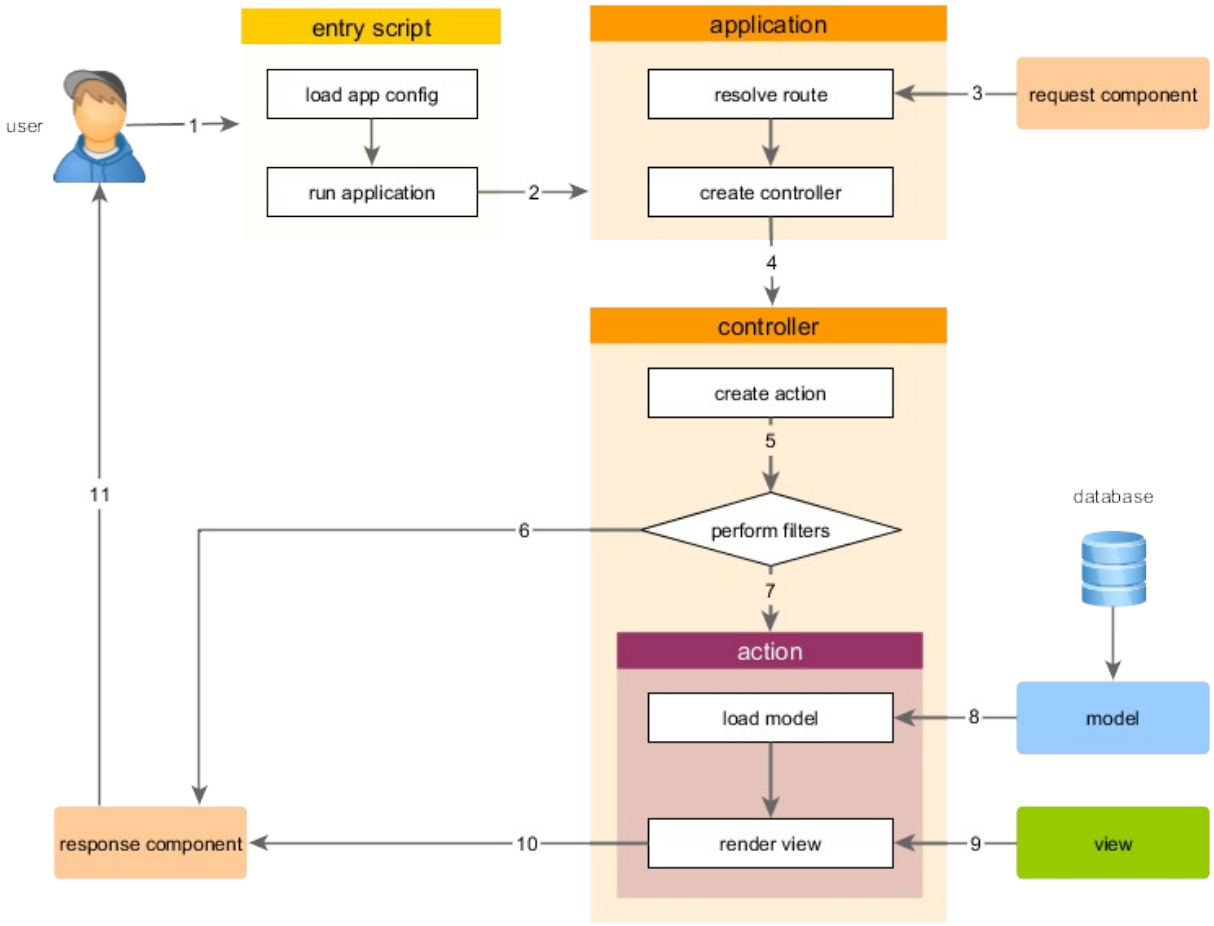
The following diagram shows the static structure of an application.



Each application has an entry script `web/index.php` which is the only Web accessible PHP script in the application. The entry script takes an incoming request and creates an application instance to handle it. The application resolves the request with the help of its components, and dispatches the request to the MVC elements. Widgets are used in the views to help build complex and dynamic user interface elements.

Request Lifecycle

The following diagram shows how an application handles a request.



1. A user makes a request to the entry script `web/index.php`.
2. The entry script loads the application configuration and creates an application instance to handle the request.
3. The application resolves the requested route with the help of the request application component.
4. The application creates a controller instance to handle the request.
5. The controller creates an action instance and performs the filters for the action.
6. If any filter fails, the action is cancelled.
7. If all filters pass, the action is executed.
8. The action loads a data model, possibly from a database.
9. The action renders a view, providing it with the data model.
10. The rendered result is returned to the response application component.
11. The response component sends the rendered result to the user's browser.

Saying Hello

This section describes how to create a new "Hello" page in your application. To achieve this goal, you will create an action and a view:

- The application will dispatch the page request to the action
- and the action will in turn render the view that shows the word "Hello" to the end user.

Through this tutorial, you will learn three things:

1. how to create an action to respond to requests,
2. how to create a view to compose the response's content, and
3. how an application dispatches requests to actions.

Creating an Action

For the "Hello" task, you will create a `say` action that reads a `message` parameter from the request and displays that message back to the user. If the request does not provide a `message` parameter, the action will display the default "Hello" message.

> Info: Actions are the objects that end users can directly refer to for execution. Actions are grouped by controllers. The execution result of an action is the response that an end user will receive.

Actions must be declared in controllers. For simplicity, you may declare the `say` action in the existing `SiteController`. This controller is defined in the class file `controllers/SiteController.php`. Here is the start of the new action:

```
<?php

namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    // ...existing code...

    public function actionSay($message = 'Hello')
    {
        return $this->render('say', ['message' => $message]);
    }
}
```

In the above code, the `say` action is defined as a method named `actionSay` in the `SiteController` class. Yii uses the prefix `action` to differentiate action methods from non-action methods in a controller class. The name after the `action` prefix maps to the action's ID.

When it comes to naming your actions, you should understand how Yii treats action IDs. Action IDs are always referenced in lower case. If an action ID requires multiple words, they will be concatenated by dashes (e.g., `create-comment`). Action method names are mapped to action IDs by removing any dashes from the IDs, capitalizing the first letter in each word, and prefixing the resulting string with `action`. For example, the action ID `create-comment` corresponds to the action method name `actionCreateComment`.

The action method in our example takes a parameter `$message`, whose value defaults to `"Hello"` (in exactly the same way you set a default value for any function or method argument in PHP). When the application receives a request and determines that the `say` action is responsible for handling said request, the application will populate this parameter with the same named parameter found in the request. In other words, if the request includes a `message` parameter with a value of `"Goodbye"`, the `$message` variable within the action will be assigned that value.

Within the action method, [[yii\web\Controller::render()|render()]] is called to render a view file named `say`. The `message` parameter is also passed to the view so that it can be used there. The rendering result is returned by the action method. That result will be received by the application and displayed to the end user in the browser (as part of a complete HTML page).

Creating a View

Views are scripts you write to generate a response's content. For the "Hello" task, you will create a `say` view that prints the `message` parameter received from the action method:

```
<?php
use yii\helpers\Html;
?>
<?= Html::encode($message) ?>
```

The `say` view should be saved in the file `views/site/say.php`. When the method [[yii\web\Controller::render()|render()]] is called in an action, it will look for a PHP file named as `views/ControllerID/ViewName.php`.

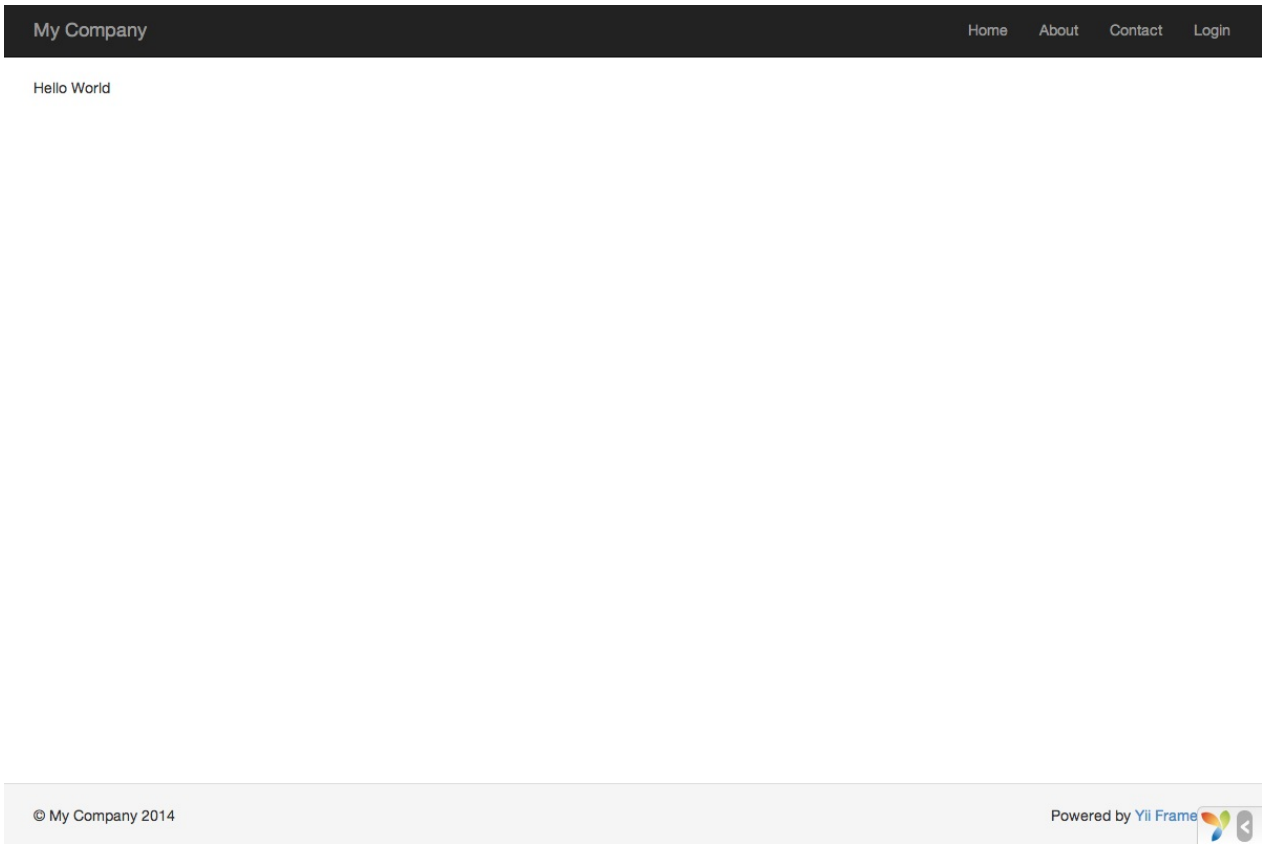
Note that in the above code, the `message` parameter is [[yii\helpers\Html::encode()|HTML-encoded]] before being printed. This is necessary as the parameter comes from an end user, making it vulnerable to cross-site scripting (XSS) attacks by embedding malicious JavaScript code in the parameter.

Naturally, you may put more content in the `say` view. The content can consist of HTML tags, plain text, and even PHP statements. In fact, the `say` view is just a PHP script that is executed by the [[yii\web\Controller::render()|render()]] method. The content printed by the view script will be returned to the application as the response's result. The application will in turn output this result to the end user.

Trying it Out

After creating the action and the view, you may access the new page by accessing the following URL:

```
http://hostname/index.php?r=site%2Fsay&message=Hello+World
```



This URL will result in a page displaying "Hello World". The page shares the same header and footer as the other application pages.

If you omit the `message` parameter in the URL, you would see the page display just "Hello". This is because `message` is passed as a parameter to the `actionSay()` method, and when it is omitted, the default value of `"Hello"` will be used instead.

> Info: The new page shares the same header and footer as other pages because the [[yii\web\Controller::render()|render()]] method will automatically embed the result of the `say` view in a so-called layout which in this case is located at `views/layouts/main.php`.

The `r` parameter in the above URL requires more explanation. It stands for route, an application wide unique ID that refers to an action. The route's format is `ControllerID/ActionID`. When the application receives a request, it will check this parameter, using the `ControllerID` part to determine which controller class should be instantiated to handle the request. Then, the controller will use the `ActionID` part to determine which action should be instantiated to do the real work. In this example case, the route `site/say` will be resolved to the `SiteController` controller class and the `say` action. As a result, the `SiteController::actionSay()` method will be called to handle the request.

> Info: Like actions, controllers also have IDs that uniquely identify them in an application. Controller IDs use the same naming rules as action IDs. Controller class names are derived from controller IDs by removing dashes from the IDs, capitalizing the first letter in each word, and suffixing the resulting string with the word `Controller`. For example, the controller ID `post-comment` corresponds to the controller class name `PostCommentController`.

Summary

In this section, you have touched the controller and view parts of the MVC architectural pattern. You created an action as part of a controller to handle a specific request. And you also created a view to compose the response's content. In this simple example, no model was involved as the only data used was the `message` parameter.

You have also learned about routes in Yii, which act as the bridge between user requests and controller actions.

In the next section, you will learn how to create a model, and add a new page containing an HTML form.

Working with Forms

This section describes how to create a new page with a form for getting data from users. The page will display a form with a name input field and an email input field. After getting those two pieces of information from the user, the page will echo the entered values back for confirmation.

To achieve this goal, besides creating an action and two views, you will also create a model.

Through this tutorial, you will learn how to:

- create a model to represent the data entered by a user through a form,
- declare rules to validate the data entered,
- build an HTML form in a view.

Creating a Model

The data to be requested from the user will be represented by an `EntryForm` model class as shown below and saved in the file `models/EntryForm.php`. Please refer to the Class Autoloading section for more details about the class file naming convention.

```
<?php

namespace app\models;

use Yii;
use yii\base\Model;

class EntryForm extends Model
{
    public $name;
    public $email;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['email', 'email'],
        ];
    }
}
```

The class extends from [[yii\base\Model]], a base class provided by Yii, commonly used to represent form data.

> Info: [[yii\base\Model]] is used as a parent for model classes *not* associated with database tables. [[yii\db\ActiveRecord]] is normally the parent for model classes that do correspond to database tables.

The `EntryForm` class contains two public members, `name` and `email`, which are used to store the data entered by the user. It also contains a method named `rules()`, which returns a set of rules for validating the data. The validation rules declared above state that

- both the `name` and `email` values are required
- the `email` data must be a syntactically valid email address

If you have an `EntryForm` object populated with the data entered by a user, you may call its [[yii\base\Model::validate()|validate()]] to trigger the data validation routines. A data validation failure will set the [[yii\base\Model::hasErrors|hasErrors]] property to `true`, and you may learn what validation errors occurred through [[yii\base\Model::getErrors|errors]].

```
<?php
$model = new EntryForm();
$model->name = 'Qiang';
$model->email = 'bad';
if ($model->validate()) {
    // Good!
} else {
    // Failure!
    // Use $model->getErrors()
}
```

Creating an Action

Next, you'll need to create an `entry` action in the `site` controller that will use the new model. The process of creating and using actions was explained in the Saying Hello section.

```php
<?php

namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\EntryForm;

class SiteController extends Controller
{
    // ...existing code...

    public function actionEntry()
    {
        $model = new EntryForm();

        if ($model->load(Yii::$app->request->post()) && $model->validate()) {
            // valid data received in $model

            // do something meaningful here about $model ...

            return $this->render('entry-confirm', ['model' => $model]);
        } else {
            // either the page is initially displayed or there is some validation error

            return $this->render('entry', ['model' => $model]);
        }
    }
}
```

The action first creates an `EntryForm` object. It then tries to populate the model with the data from `$_POST`, provided in Yii by [[yii\web\Request::post()]]. If the model is successfully populated (i.e., if the user has submitted the HTML form), the action will call [[yii\base\Model::validate()|validate()]] to make sure the values entered are valid.

> Info: The expression `Yii::$app` represents the application instance, which is a globally accessible singleton. It is also a service locator that provides components such as `request`, `response`, `db`, etc. to support specific functionality. In the above code, the `request` component of the application instance is used to access the `$_POST` data.

If everything is fine, the action will render a view named `entry-confirm` to confirm the successful submission of the data to the user. If no data is submitted or the data contains errors, the `entry` view will be rendered, wherein the HTML form will be shown, along with any validation error messages.

> Note: In this very simple example we just render the confirmation page upon valid data submission. In practice, you should consider using [[yii\web\Controller::refresh()|refresh()]] or [[yii\web\Controller::redirect()|redirect()]] to avoid form resubmission problems.

Creating Views

Finally, create two view files named `entry-confirm` and `entry`. These will be rendered by the `entry` action, as just described.

The `entry-confirm` view simply displays the name and email data. It should be stored in the file `views/site/entry-confirm.php`.

```
<?php
use yii\helpers\Html;
?>
<p>You have entered the following information:</p>

<ul>
    <li><label>Name</label>: <?= Html::encode($model->name) ?></li>
    <li><label>Email</label>: <?= Html::encode($model->email) ?></li>
</ul>
```

The `entry` view displays an HTML form. It should be stored in the file `views/site/entry.php`.

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;
?>
<?php $form = ActiveForm::begin(); ?>

    <?= $form->field($model, 'name') ?>

    <?= $form->field($model, 'email') ?>

    <div class="form-group">
        <?= Html::submitButton('Submit', ['class' => 'btn btn-primary']) ?>
    </div>

<?php ActiveForm::end(); ?>
```

The view uses a powerful widget called [[yii\widgets\ActiveForm|ActiveForm]] to build the HTML form. The `begin()` and `end()` methods of the widget render the opening and closing form tags, respectively. Between the two method calls, input fields are created by the

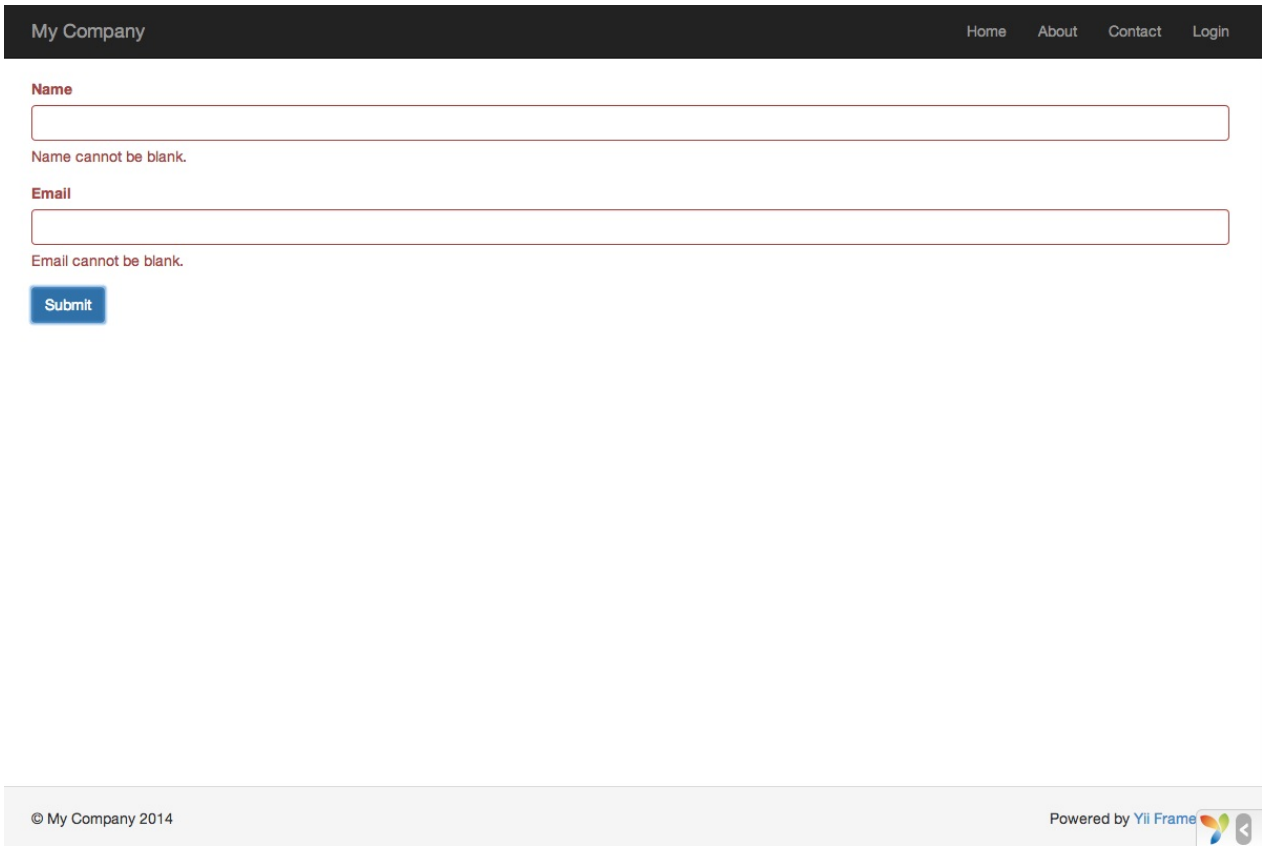
[[yii\widgets\ActiveForm::field()|field()]] method. The first input field is for the "name" data, and the second for the "email" data. After the input fields, the [[yii\helpers\Html::submitButton()]] method is called to generate a submit button.

Trying it Out

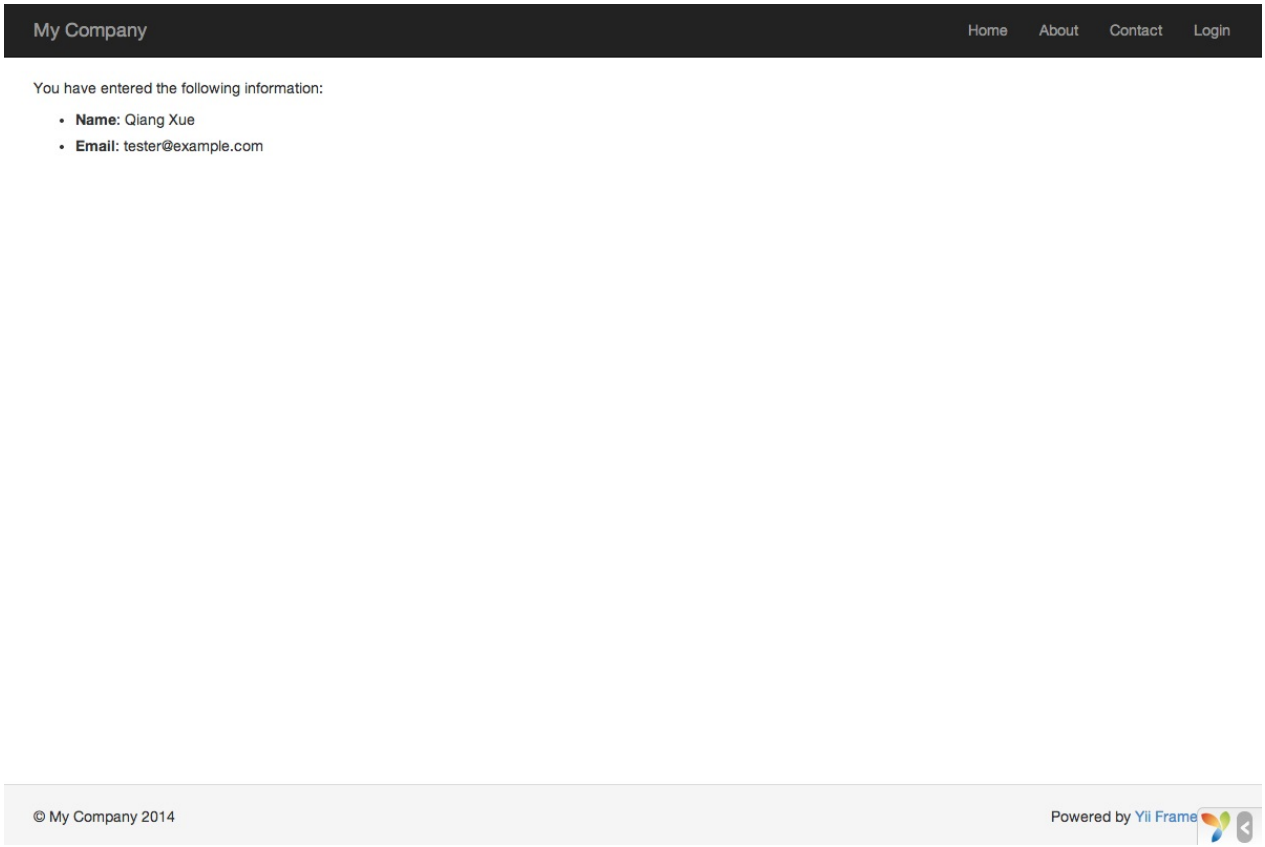
To see how it works, use your browser to access the following URL:

```
http://hostname/index.php?r=site%2Fentry
```

You will see a page displaying a form with two input fields. In front of each input field, a label indicates what data is to be entered. If you click the submit button without entering anything, or if you do not provide a valid email address, you will see an error message displayed next to each problematic input field.



After entering a valid name and email address and clicking the submit button, you will see a new page displaying the data that you just entered.



Magic Explained

You may wonder how the HTML form works behind the scene, because it seems almost magical that it can display a label for each input field and show error messages if you do not enter the data correctly without reloading the page.

Yes, the data validation is initially done on the client-side using JavaScript, and secondarily performed on the server-side via PHP. [[yii\widgets\ActiveForm]] is smart enough to extract the validation rules that you have declared in `EntryForm`, turn them into executable JavaScript code, and use the JavaScript to perform data validation. In case you have disabled JavaScript on your browser, the validation will still be performed on the server-side, as shown in the `actionEntry()` method. This ensures data validity in all circumstances.

> Warning: Client-side validation is a convenience that provides for a better user experience. Server-side validation is always required, whether or not client-side validation is in place.

The labels for input fields are generated by the `field()` method, using the property names from the model. For example, the label `Name` will be generated for the `name` property.

You may customize a label within a view using the following code:

```
<?= $form->field($model, 'name')->label('Your Name') ?>
<?= $form->field($model, 'email')->label('Your Email') ?>
```

> Info: Yii provides many such widgets to help you quickly build complex and dynamic views. As you will learn later, writing a new widget is also extremely easy. You may want to turn much of your view code into reusable widgets to simplify view development in future.

Summary

In this section of the guide, you have touched every part in the MVC architectural pattern. You have learned how to create a model class to represent the user data and validate said data.

You have also learned how to get data from users and how to display data back in the browser. This is a task that could take you a lot of time when developing an application, but Yii provides powerful widgets to make this task very easy.

In the next section, you will learn how to work with databases, which are needed in nearly every application.

Working with Databases

This section will describe how to create a new page that displays country data fetched from a database table named `country`. To achieve this goal, you will configure a database connection, create an Active Record class, define an action, and create a view.

Through this tutorial, you will learn how to:

- configure a DB connection,
- define an Active Record class,
- query data using the Active Record class,
- display data in a view in a paginated fashion.

Note that in order to finish this section, you should have basic knowledge and experience using databases. In particular, you should know how to create a database, and how to execute SQL statements using a DB client tool.

Preparing the Database

To begin, create a database named `yii2basic`, from which you will fetch data in your application. You may create an SQLite, MySQL, PostgreSQL, MSSQL or Oracle database, as Yii has built-in support for many database applications. For simplicity, MySQL will be assumed in the following description.

Next, create a table named `country` in the database, and insert some sample data. You may run the following SQL statements to do so:

```
CREATE TABLE `country` (
  `code` CHAR(2) NOT NULL PRIMARY KEY,
  `name` CHAR(52) NOT NULL,
  `population` INT(11) NOT NULL DEFAULT '0'
) ENGINE=InnoDB DEFAULT CHARSET=utf8;

INSERT INTO `country` VALUES ('AU','Australia',24016400);
INSERT INTO `country` VALUES ('BR','Brazil',205722000);
INSERT INTO `country` VALUES ('CA','Canada',35985751);
INSERT INTO `country` VALUES ('CN','China',1375210000);
INSERT INTO `country` VALUES ('DE','Germany',81459000);
INSERT INTO `country` VALUES ('FR','France',64513242);
INSERT INTO `country` VALUES ('GB','United Kingdom',65097000);
INSERT INTO `country` VALUES ('IN','India',1285400000);
INSERT INTO `country` VALUES ('RU','Russia',146519759);
INSERT INTO `country` VALUES ('US','United States',322976000);
```

At this point, you have a database named `yii2basic`, and within it a `country` table with three columns, containing ten rows of data.

Configuring a DB Connection

Before proceeding, make sure you have installed both the PDO PHP extension and the PDO driver for the database you are using (e.g. `pdo_mysql` for MySQL). This is a basic requirement if your application uses a relational database.

With those installed, open the file `config/db.php` and change the parameters to be correct for your database. By default, the file contains the following:

```
<?php

return [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=yii2basic',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];
```

The `config/db.php` file is a typical file-based configuration tool. This particular configuration file specifies the parameters needed to create and initialize a [[yii\db\Connection]] instance through which you can make SQL queries against the underlying database.

The DB connection configured above can be accessed in the application code via the expression `Yii::$app->db`.

> Info: The `config/db.php` file will be included by the main application configuration `config/web.php`, which specifies how the application instance should be initialized. For more information, please refer to the Configurations section.

If you need to work with databases support for which isn't bundled with Yii, check the following extensions:

- Informix
- IBM DB2
- Firebird

Creating an Active Record

To represent and fetch the data in the `country` table, create an Active Record-derived class named `Country`, and save it in the file `models/Country.php`.

```
<?php

namespace app\models;

use yii\db\ActiveRecord;

class Country extends ActiveRecord
{
}
```

The `Country` class extends from [[yii\db\ActiveRecord]]. You do not need to write any code inside of it! With just the above code, Yii will guess the associated table name from the class name.

> Info: If no direct match can be made from the class name to the table name, you can override the [[yii\db\ActiveRecord::tableName()]] method to explicitly specify the associated table name.

Using the `Country` class, you can easily manipulate data in the `country` table, as shown in these snippets:

```
use app\models\Country;

// get all rows from the country table and order them by "name"
$countries = Country::find()->orderBy('name')->all();

// get the row whose primary key is "US"
$country = Country::findOne('US');

// displays "United States"
echo $country->name;

// modifies the country name to be "U.S.A." and save it to database
$country->name = 'U.S.A.';
$country->save();
```

> Info: Active Record is a powerful way to access and manipulate database data in an object-oriented fashion. You may find more detailed information in the Active Record section. Alternatively, you may also interact with a database using a lower-level data accessing method called Data Access Objects.

Creating an Action

To expose the country data to end users, you need to create a new action. Instead of placing the new action in the `site` controller, like you did in the previous sections, it makes more sense to create a new controller specifically for all actions related to the country data. Name this new controller `CountryController`, and create an `index` action in it, as shown in the following.

```
<?php

namespace app\controllers;

use yii\web\Controller;
use yii\data\Pagination;
use app\models\Country;

class CountryController extends Controller
{
    public function actionIndex()
    {
        $query = Country::find();

        $pagination = new Pagination([
            'defaultPageSize' => 5,
            'totalCount' => $query->count(),
        ]);

        $countries = $query->orderBy('name')
            ->offset($pagination->offset)
            ->limit($pagination->limit)
            ->all();

        return $this->render('index', [
            'countries' => $countries,
            'pagination' => $pagination,
        ]);
    }
}
```

Save the above code in the file `controllers/CountryController.php`.

The `index` action calls `Country::find()`. This Active Record method builds a DB query and retrieves all of the data from the `country` table. To limit the number of countries returned in each request, the query is paginated with the help of a [[yii\data\Pagination]] object. The `Pagination` object serves two purposes:

- Sets the `offset` and `limit` clauses for the SQL statement represented by the query so that it only returns a single page of data at a time (at most 5 rows in a page).
- It's used in the view to display a pager consisting of a list of page buttons, as will be explained in the next subsection.

At the end of the code, the `index` action renders a view named `index`, and passes the country data as well as the pagination information to it.

Creating a View

Under the `views` directory, first create a sub-directory named `country`. This folder will be used to hold all the views rendered by the `country` controller. Within the `views/country` directory, create a file named `index.php` containing the following:

```
<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>
<h1>Countries</h1>
<ul>
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->name} ({$country->code})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

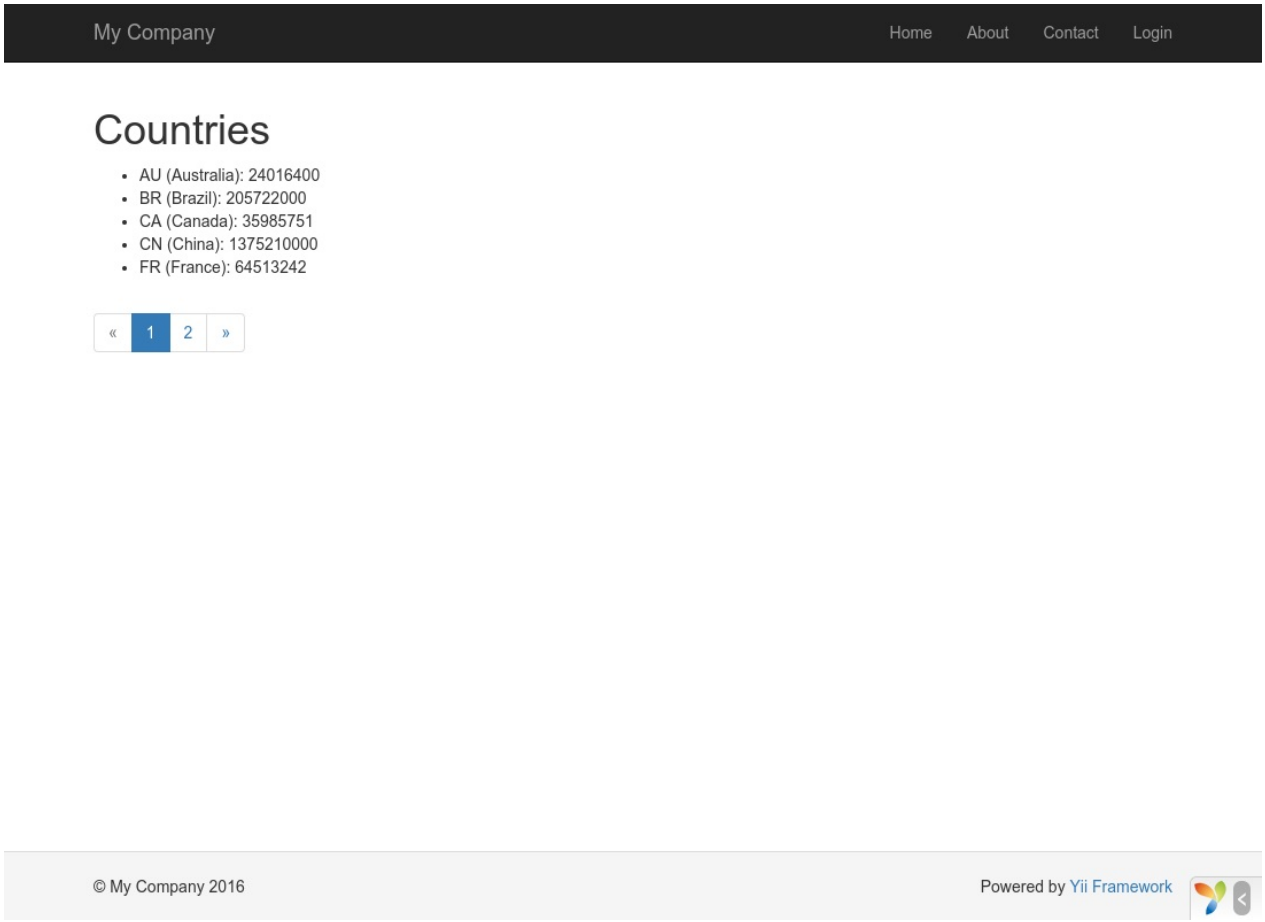
<?= LinkPager::widget(['pagination' => $pagination]) ?>
```

The view has two sections relative to displaying the country data. In the first part, the provided country data is traversed and rendered as an unordered HTML list. In the second part, a [[yii\widgets\LinkPager]] widget is rendered using the pagination information passed from the action. The `LinkPager` widget displays a list of page buttons. Clicking on any of them will refresh the country data in the corresponding page.

Trying it Out

To see how all of the above code works, use your browser to access the following URL:

```
http://hostname/index.php?r=country%2Findex
```



At first, you will see a page showing five countries. Below the countries, you will see a pager with four buttons. If you click on the button "2", you will see the page display another five countries in the database: the second page of records. Observe more carefully and you will find that the URL in the browser also changes to

```
http://hostname/index.php?r=country%2Findex&page=2
```

Behind the scenes, [[yii\data\Pagination|Pagination]] is providing all of the necessary functionality to paginate a data set:

- Initially, [[yii\data\Pagination|Pagination]] represents the first page, which reflects the country SELECT query with the clause `LIMIT 5 OFFSET 0`. As a result, the first five countries will be fetched and displayed.
- The [[yii\widgets\LinkPager|LinkPager]] widget renders the page buttons using the URLs created by [[yii\data\Pagination::createUrl()|Pagination]]. The URLs will contain the query parameter `page`, which represents the different page numbers.
- If you click the page button "2", a new request for the route `country/index` will be triggered and handled. [[yii\data\Pagination|Pagination]] reads the `page` query parameter from the URL and sets the current page number to 2. The new country query will thus have the clause `LIMIT 5 OFFSET 5` and return the next five countries for display.

Summary

In this section, you learned how to work with a database. You also learned how to fetch and display data in pages with the help of [[yii\data\Pagination]] and [[yii\widgets\LinkPager]].

In the next section, you will learn how to use the powerful code generation tool, called Gii, to help you rapidly implement some commonly required features, such as the Create-Read-Update-Delete (CRUD) operations for working with the data in a database table. As a matter of fact, the code you have just written can all be automatically generated in Yii using the Gii tool.

Generating Code with Gii

This section will describe how to use Gii to automatically generate code that implements some common Web site features. Using Gii to auto-generate code is simply a matter of entering the right information per the instructions shown on the Gii Web pages.

Through this tutorial, you will learn how to:

- enable Gii in your application,
- use Gii to generate an Active Record class,
- use Gii to generate the code implementing the CRUD operations for a DB table,
- customize the code generated by Gii.

Starting Gii

Gii is provided in Yii as a module. You can enable Gii by configuring it in the [[yii\base\Application::modules|modules]] property of the application. Depending upon how you created your application, you may find the following code is already provided in the `config/web.php` configuration file:

```
$config = [ ... ];

if (YII_ENV_DEV) {
    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
    ];
}
```

The above configuration states that when in development environment, the application should include a module named `gii`, which is of class [[yii\gii\Module]].

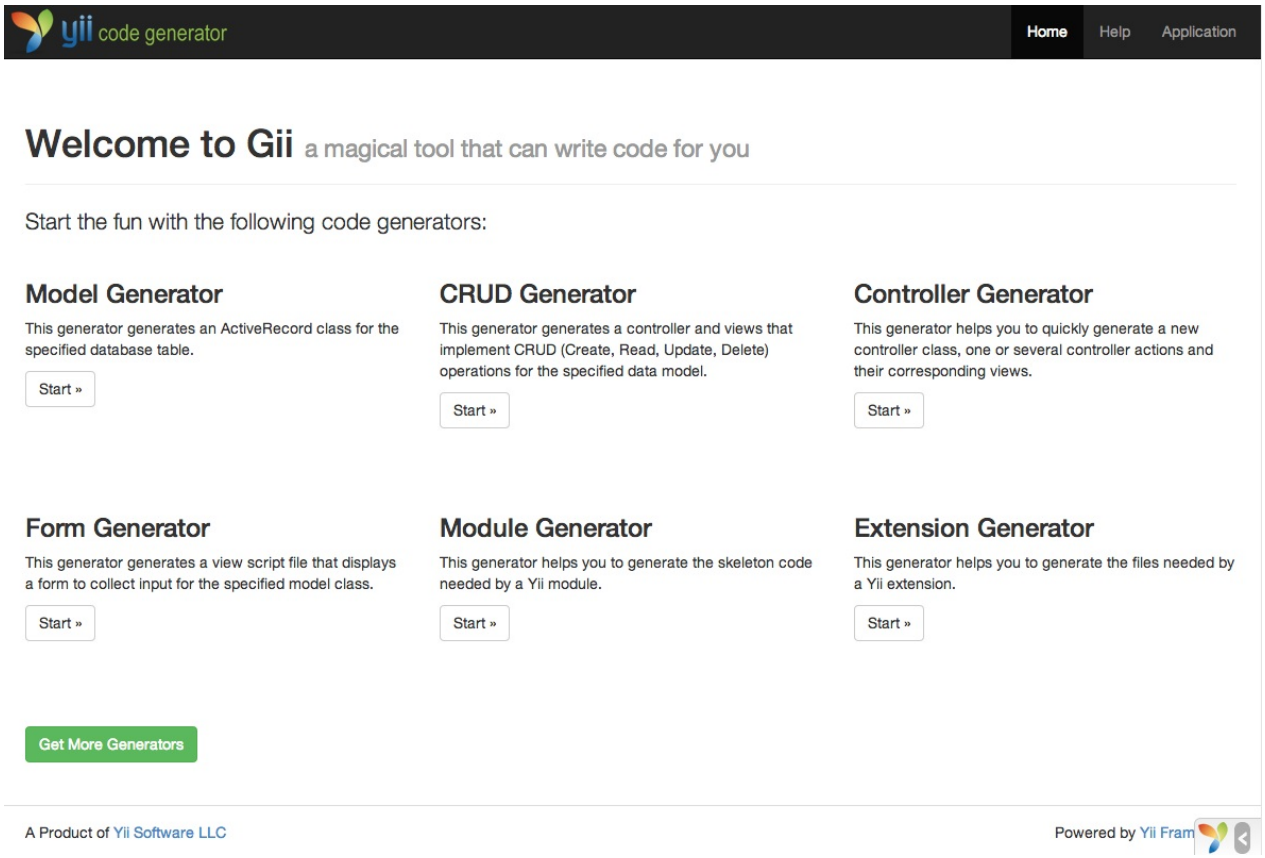
If you check the entry script `web/index.php` of your application, you will find the following line, which essentially makes `YII_ENV_DEV` to be `true`.

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

Thanks to that line, your application is in development mode, and will have already enabled Gii, per the above configuration. You can now access Gii via the following URL:

```
http://hostname/index.php?r=gii
```

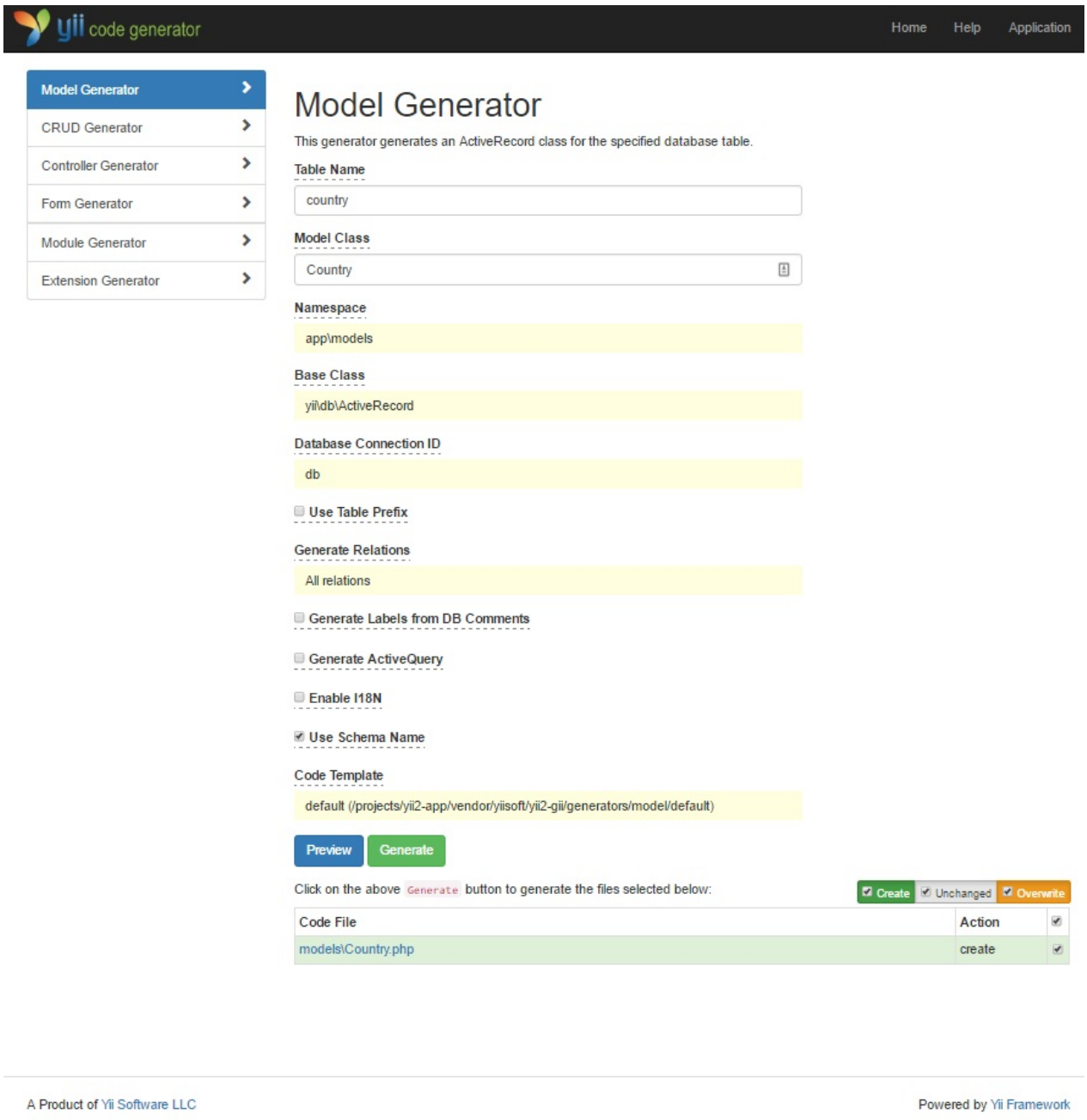
> Note: If you are accessing Gii from a machine other than localhost, the access will be denied by default for security purpose. You can configure Gii to add the allowed IP addresses as follows,
>
> ```
> 'gii' => [
>     'class' => 'yii\gii\Module',
>     'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '192.168.178.20'] // adju
> st this to your needs
> ],
> ```



Generating an Active Record Class

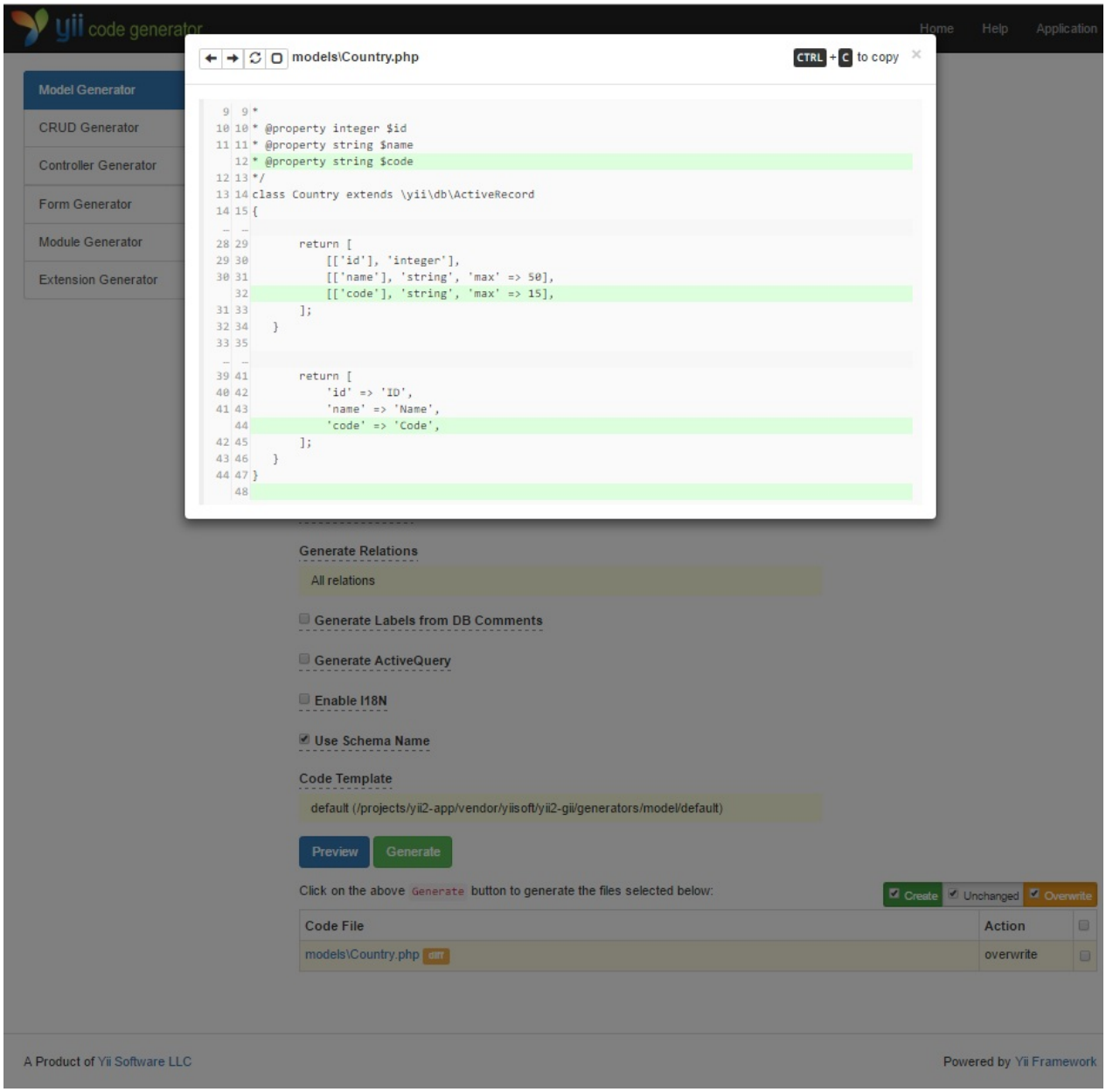
To use Gii to generate an Active Record class, select the "Model Generator" (by clicking the link on the Gii index page). Then fill out the form as follows:

- Table Name: `country`
- Model Class: `Country`



Next, click on the "Preview" button. You will see `models/Country.php` is listed in the resulting class file to be created. You may click on the name of the class file to preview its content.

When using Gii, if you have already created the same file and would be overwriting it, click the `diff` button next to the file name to see the differences between the code to be generated and the existing version.



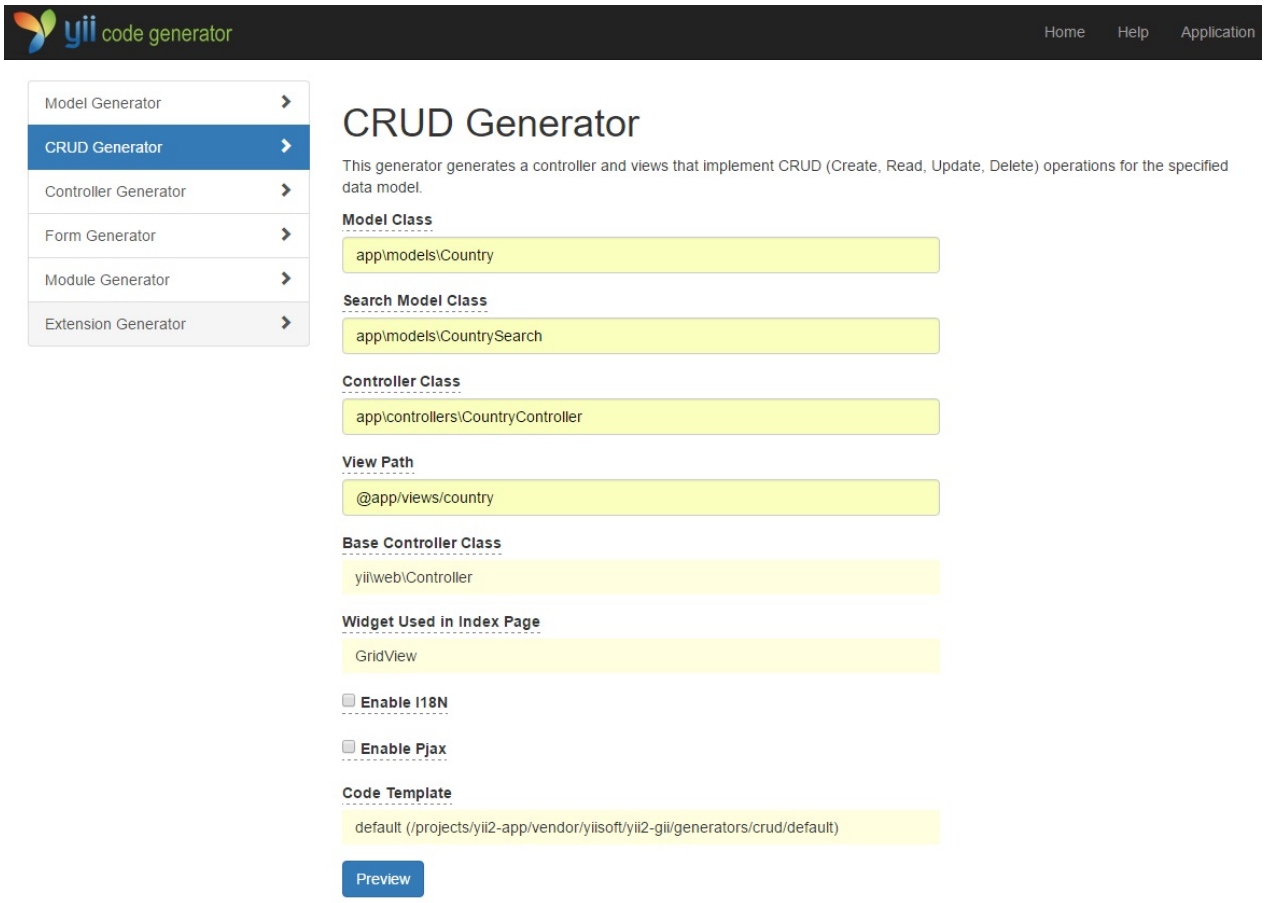
When overwriting an existing file, check the box next to "overwrite" and then click the "Generate" button. If creating a new file, you can just click "Generate".

Next, you will see a confirmation page indicating the code has been successfully generated. If you had an existing file, you'll also see a message indicating that it was overwritten with the newly generated code.

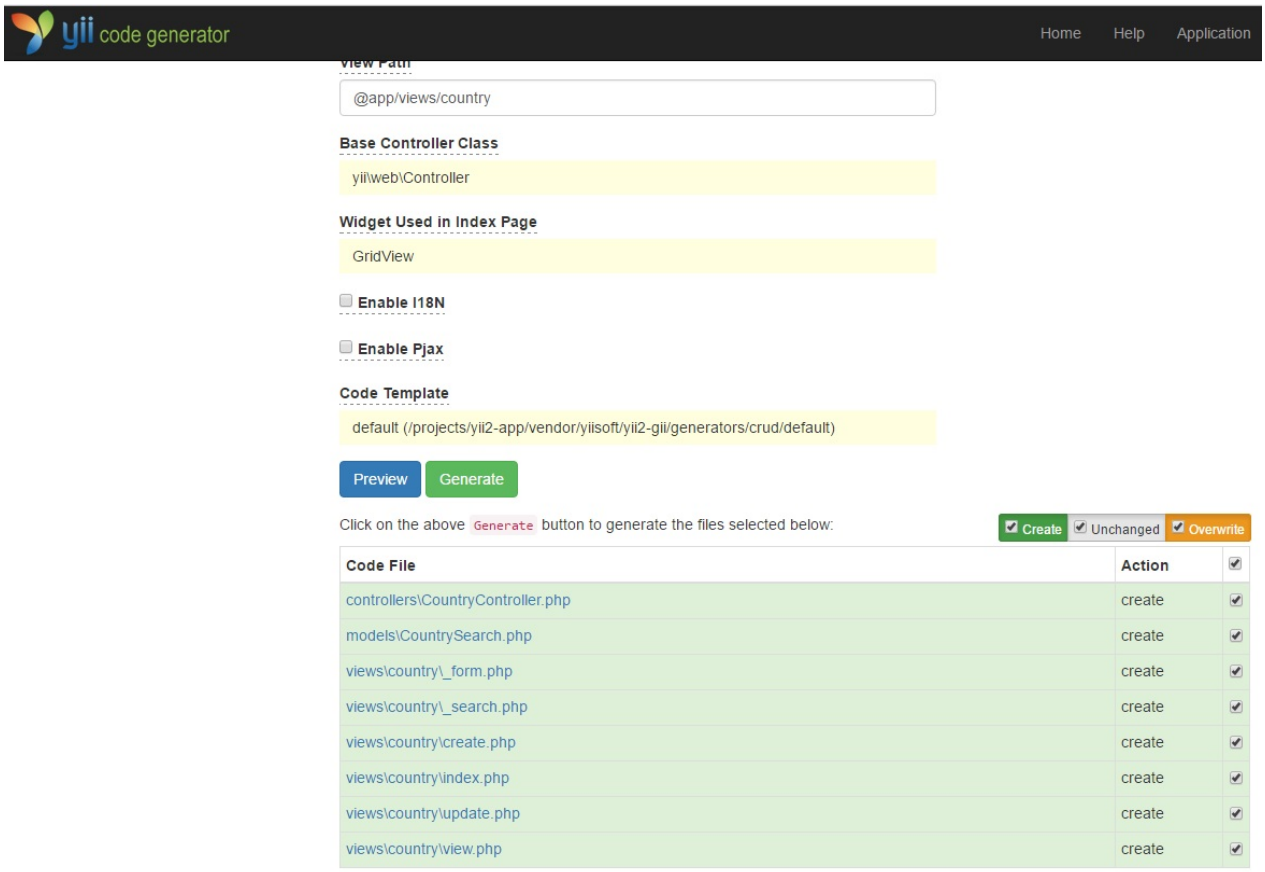
Generating CRUD Code

CRUD stands for Create, Read, Update, and Delete, representing the four common tasks taken with data on most Web sites. To create CRUD functionality using Gii, select the "CRUD Generator" (by clicking the link on the Gii index page). For the "country" example, fill out the resulting form as follows:

- Model Class: `app\models\Country`
- Search Model Class: `app\models\CountrySearch`
- Controller Class: `app\controllers\CountryController`



Next, click on the "Preview" button. You will see a list of files to be generated, as shown below.



If you previously created the `controllers/CountryController.php` and `views/country/index.php` files (in the databases section of the guide), check the "overwrite" box to replace them. (The previous versions did not have full CRUD support.)

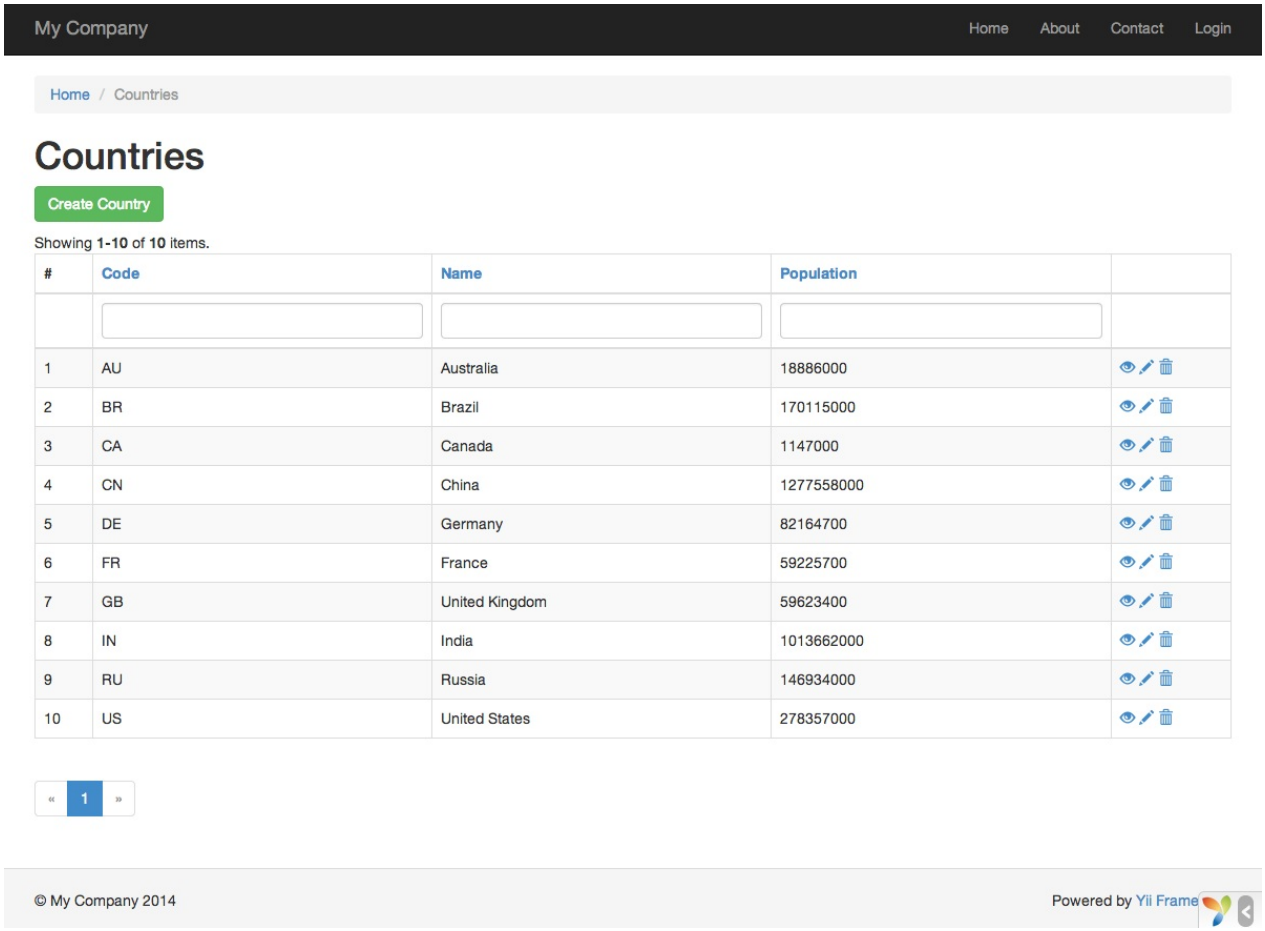
Trying it Out

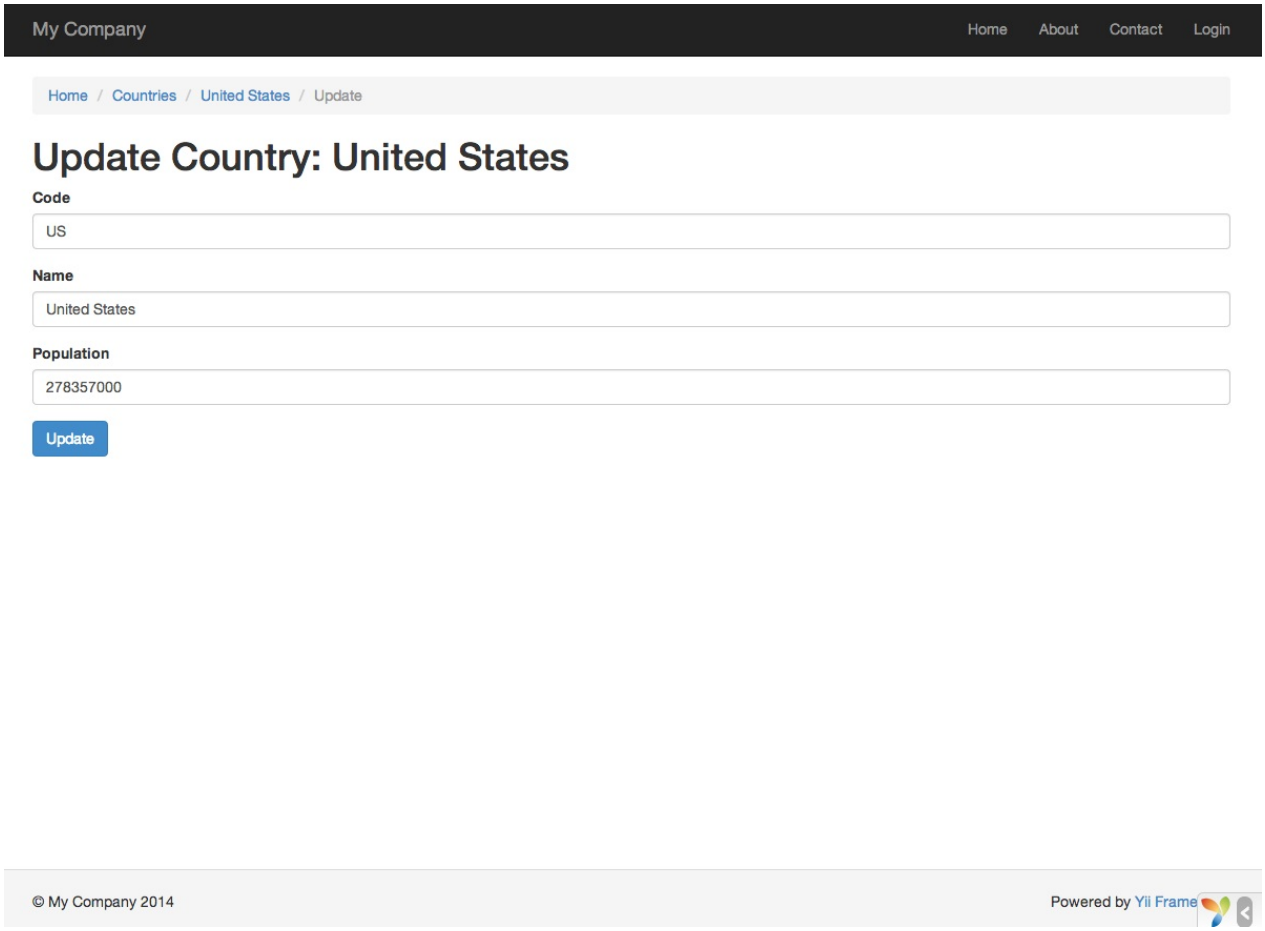
To see how it works, use your browser to access the following URL:

```
http://hostname/index.php?r=country%2Findex
```

You will see a data grid showing the countries from the database table. You may sort the grid, or filter it by entering filter conditions in the column headers.

For each country displayed in the grid, you may choose to view its details, update it, or delete it. You may also click on the "Create Country" button on top of the grid to be provided with a form for creating a new country.


My Company
Home
About
Contact
Login
Home / Countries
Countries
Create Country
Showing 1-10 of 10 items.
Code Name Population
1 AU Australia 18886000
2 BR Brazil 170115000
3 CA Canada 1147000
4 CN China 1277558000
5 DE Germany 82164700
6 FR France 59225700
7 GB United Kingdom 59623400
8 IN India 1013662000
9 RU Russia 146934000
10 US United States 278357000
« 1 »
© My Company 2014
Powered by Yii Frame


My Company
Home
About
Contact
Login
Home / Countries / United States / Update
Update Country: United States
Code
US
Name
United States
Population
278357000
Update
© My Company 2014
Powered by Yii Frame

The following is the list of the files generated by Gii, in case you want to investigate how these features are implemented, or to customize them:

- Controller: `controllers/CountryController.php`
- Models: `models/Country.php` and `models/CountrySearch.php`
- Views: `views/country/*.php`

> Info: Gii is designed to be a highly customizable and extensible code generation tool. Using it wisely can greatly accelerate your application development speed. For more details, please refer to the Gii section.

Summary

In this section, you have learned how to use Gii to generate the code that implements complete CRUD functionality for content stored in a database table.

Looking Ahead

If you've read through the entire "Getting Started" chapter, you have now created a complete Yii application. In the process, you have learned how to implement some commonly needed features, such as getting data from users via an HTML form, fetching data from a database, and displaying data in a paginated fashion. You have also learned how to use Gii to generate code automatically. Using Gii for code generation turns the bulk of your Web development process into a task as simple as just filling out some forms.

This section will summarize the Yii resources available to help you be more productive when using the framework.

- Documentation
  - The Definitive Guide: As the name indicates, the guide precisely defines how Yii should work and provides general guidance about using Yii. It is the single most important Yii tutorial, and one that you should read before writing any Yii code.
  - The Class Reference: This specifies the usage of every class provided by Yii. It should be mainly used when you are writing code and want to understand the usage of a particular class, method, property. Usage of the class reference is best only after a contextual understanding of the entire framework.
  - The Wiki Articles: The wiki articles are written by Yii users based on their own experiences. Most of them are written like cookbook recipes, and show how to solve particular problems using Yii. While the quality of these articles may not be as good as the Definitive Guide, they are useful in that they cover broader topics and can often provide ready-to-use solutions.
  - Books
- Extensions: Yii boasts a library of thousands of user-contributed extensions that can be easily plugged into your applications, thereby making your application development even faster and easier.
- Community
  - Forum: http://www.yiiframework.com/forum/
  - IRC chat: The #yii channel on the freenode network (irc://irc.freenode.net/yii)
  - Gitter chat: https://gitter.im/yiisoft/yii2
  - GitHub: https://github.com/yiisoft/yii2
  - Facebook: https://www.facebook.com/groups/yiitalk/
  - Twitter: https://twitter.com/yiiframework
  - LinkedIn: https://www.linkedin.com/groups/yii-framework-1483367
  - Stackoverflow: http://stackoverflow.com/questions/tagged/yii2

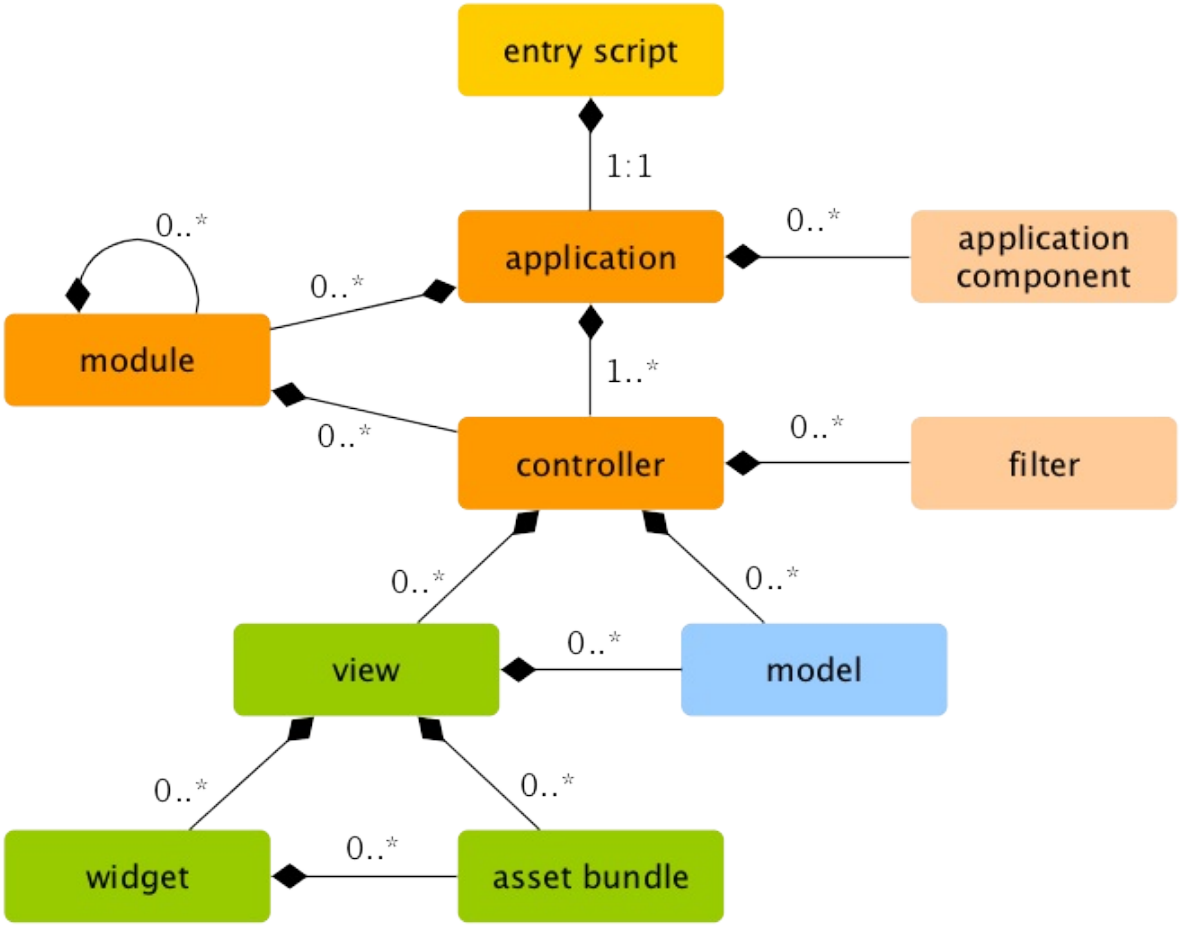
Overview

Yii applications are organized according to the model-view-controller (MVC) architectural pattern. Models represent data, business logic and rules; views are output representation of models; and controllers take input and convert it to commands for models and views.

Besides MVC, Yii applications also have the following entities:

- entry scripts: they are PHP scripts that are directly accessible by end users. They are responsible for starting a request handling cycle.
- applications: they are globally accessible objects that manage application components and coordinate them to fulfill requests.
- application components: they are objects registered with applications and provide various services for fulfilling requests.
- modules: they are self-contained packages that contain complete MVC by themselves. An application can be organized in terms of multiple modules.
- filters: they represent code that need to be invoked before and after the actual handling of each request by controllers.
- widgets: they are objects that can be embedded in views. They may contain controller logic and can be reused in different views.

The following diagram shows the static structure of an application:


entry script
1:1
application
0..*
application component
0..*
module
0..*
1..*
0..*
controller
0..*
filter
0..*
0..*
view
0..*
model
0..*
0..*
widget
0..*
asset bundle

Entry Scripts

Entry scripts are the first step in the application bootstrapping process. An application (either Web application or console application) has a single entry script. End users make requests to entry scripts which instantiate application instances and forward the requests to them.

Entry scripts for Web applications must be stored under Web accessible directories so that they can be accessed by end users. They are often named as `index.php`, but can also use any other names, provided Web servers can locate them.

Entry scripts for console applications are usually stored under the base path of applications and are named as `yii` (with the `.php` suffix). They should be made executable so that users can run console applications through the command `./yii <route> [arguments] [options]`.

Entry scripts mainly do the following work:

- Define global constants;
- Register Composer autoloader;
- Include the [[Yii]] class file;
- Load application configuration;
- Create and configure an application instance;
- Call [[yii\base\Application::run()]] to process the incoming request.

Web Applications

The following is the code in the entry script for the Basic Web Project Template.

```php
<?php

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

// register Composer autoloader
require(__DIR__ . '/../vendor/autoload.php');

// include Yii class file
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

// load application configuration
$config = require(__DIR__ . '/../config/web.php');

// create, configure and run application
(new yii\web\Application($config))->run();
```

Console Applications

Similarly, the following is the code for the entry script of a console application:

```php
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 *
 * @link http://www.yiiframework.com/
 * @copyright Copyright (c) 2008 Yii Software LLC
 * @license http://www.yiiframework.com/license/
 */

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

// register Composer autoloader
require(__DIR__ . '/vendor/autoload.php');

// include Yii class file
require(__DIR__ . '/vendor/yiisoft/yii2/Yii.php');

// load application configuration
$config = require(__DIR__ . '/config/console.php');

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

Defining Constants

Entry scripts are the best place for defining global constants. Yii supports the following three constants:

- `YII_DEBUG`: specifies whether the application is running in debug mode. When in debug mode, an application will keep more log information, and will reveal detailed error call stacks if exceptions are thrown. For this reason, debug mode should be used mainly during development. The default value of `YII_DEBUG` is `false`.
- `YII_ENV`: specifies which environment the application is running in. This will be described in more detail in the Configurations section. The default value of `YII_ENV` is `'prod'`, meaning the application is running in production environment.
- `YII_ENABLE_ERROR_HANDLER`: specifies whether to enable the error handler provided by Yii. The default value of this constant is `true`.

When defining a constant, we often use the code like the following:

```
defined('YII_DEBUG') or define('YII_DEBUG', true);
```

which is equivalent to the following code:

```
if (!defined('YII_DEBUG')) {
    define('YII_DEBUG', true);
}
```

Clearly the former is more succinct and easier to understand.

Constant definitions should be done at the very beginning of an entry script so that they can take effect when other PHP files are being included.

Applications

Applications are objects that govern the overall structure and lifecycle of Yii application systems. Each Yii application system contains a single application object which is created in the entry script and is globally accessible through the expression `\Yii::$app`.

> Info: Depending on the context, when we say "an application", it can mean either an application object or an application system.

There are two types of applications: [[yii\web\Application|Web applications]] and [[yii\console\Application|console applications]]. As the names indicate, the former mainly handles Web requests, while the latter handles console command requests.

Application Configurations

When an entry script creates an application, it will load a configuration and apply it to the application, as follows:

```
require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

// load application configuration
$config = require(__DIR__ . '/../config/web.php');

// instantiate and configure the application
(new yii\web\Application($config))->run();
```

Like normal configurations, application configurations specify how to initialize properties of application objects. Because application configurations are often very complex, they usually are kept in configuration files, like the `web.php` file in the above example.

Application Properties

There are many important application properties that you should configure in application configurations. These properties typically describe the environment that applications are running in. For example, applications need to know how to load controllers, where to store temporary files, etc. In the following, we will summarize these properties.

Required Properties

In any application, you should at least configure two properties: [[yii\base\Application::id|id]] and [[yii\base\Application::basePath|basePath]].

[[yii\base\Application::id|id]]

The [[yii\base\Application::id|id]] property specifies a unique ID that differentiates an application from others. It is mainly used programmatically. Although not a requirement, for best interoperability it is recommended that you use only alphanumeric characters when specifying an application ID.

[[yii\base\Application::basePath|basePath]]

The [[yii\base\Application::basePath|basePath]] property specifies the root directory of an application. It is the directory that contains all protected source code of an application system. Under this directory, you normally will see sub-directories such as `models`, `views`, and `controllers`, which contain source code corresponding to the MVC pattern.

You may configure the [[yii\base\Application::basePath|basePath]] property using a directory path or a path alias. In both forms, the corresponding directory must exist, or an exception will be thrown. The path will be normalized by calling the `realpath()` function.

The [[yii\base\Application::basePath|basePath]] property is often used to derive other important paths (e.g. the runtime path). For this reason, a path alias named `@app` is predefined to represent this path. Derived paths may then be formed using this alias (e.g. `@app/runtime` to refer to the runtime directory).

Important Properties

The properties described in this subsection often need to be configured because they differ across different applications.

[[yii\base\Application::aliases|aliases]]

This property allows you to define a set of aliases in terms of an array. The array keys are alias names, and the array values are the corresponding path definitions. For example:

```
[
    'aliases' => [
        '@name1' => 'path/to/path1',
        '@name2' => 'path/to/path2',
    ],
]
```

This property is provided so that you can define aliases in terms of application configurations instead of by calling the [[Yii::setAlias()]] method.

[[yii\base\Application::bootstrap|bootstrap]]

This is a very useful property. It allows you to specify an array of components that should be run during the application [[yii\base\Application::bootstrap()|bootstrapping process]]. For example, if you want a module to customize the URL rules, you may list its ID as an element in this property.

Each component listed in this property may be specified in one of the following formats:

- an application component ID as specified via components,
- a module ID as specified via modules,
- a class name,
- a configuration array,
- an anonymous function that creates and returns a component.

For example:

```
[
    'bootstrap' => [
        // an application component ID or module ID
        'demo',

        // a class name
        'app\components\Profiler',

        // a configuration array
        [
            'class' => 'app\components\Profiler',
            'level' => 3,
        ],

        // an anonymous function
        function () {
            return new app\components\Profiler();
        }
    ],
]
```

> Info: If a module ID is the same as an application component ID, the application component will be used during the bootstrapping process. If you want to use the module instead, you may specify it using an anonymous function like the following:
>
> ```
> [
>     function () {
>         return Yii::$app->getModule('user');
>     },
> ]
> ```

During the bootstrapping process, each component will be instantiated. If the component class implements [[yii\base\BootstrapInterface]], its [[yii\base\BootstrapInterface::bootstrap()|bootstrap()]] method will also be called.

Another practical example is in the application configuration for the Basic Project Template, where the `debug` and `gii` modules are configured as bootstrapping components when the application is running in the development environment:

```
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = 'yii\gii\Module';
}
```

> Note: Putting too many components in `bootstrap` will degrade the performance of your application because for each request, the same set of components need to be run. So use bootstrapping components judiciously.

[[yii\web\Application::catchAll|catchAll]]

This property is supported by [[yii\web\Application|Web applications]] only. It specifies a controller action which should handle all user requests. This is mainly used when the application is in maintenance mode and needs to handle all incoming requests via a single action.

The configuration is an array whose first element specifies the route of the action. The rest of the array elements (key-value pairs) specify the parameters to be bound to the action. For example:

```
[
    'catchAll' => [
        'offline/notice',
        'param1' => 'value1',
        'param2' => 'value2',
    ],
]
```

> Info: Debug panel on development environment will not work when this property is enabled.

[[yii\base\Application::components|components]]

This is the single most important property. It allows you to register a list of named components called application components that you can use in other places. For example:

```
[
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
        ],
    ],
]
```

Each application component is specified as a key-value pair in the array. The key represents the component ID, while the value represents the component class name or configuration.

You can register any component with an application, and the component can later be accessed globally using the expression `\Yii::$app->componentID`.

Please read the Application Components section for details.

[[yii\base\Application::controllerMap|controllerMap]]

This property allows you to map a controller ID to an arbitrary controller class. By default, Yii maps controller IDs to controller classes based on a convention (e.g. the ID `post` would be mapped to `app\controllers\PostController`). By configuring this property, you can break the convention for specific controllers. In the following example, `account` will be mapped to `app\controllers\UserController`, while `article` will be mapped to `app\controllers\PostController`.

```
[
    'controllerMap' => [
        'account' => 'app\controllers\UserController',
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

The array keys of this property represent the controller IDs, while the array values represent the corresponding controller class names or configurations.

[[yii\base\Application::controllerNamespace|controllerNamespace]]

This property specifies the default namespace under which controller classes should be located. It defaults to `app\controllers`. If a controller ID is `post`, by convention the corresponding controller class name (without namespace) would be `PostController`, and the fully qualified class name would be `app\controllers\PostController`.

Controller classes may also be located under sub-directories of the directory corresponding to this namespace. For example, given a controller ID `admin/post`, the corresponding fully qualified controller class would be `app\controllers\admin\PostController`.

It is important that the fully qualified controller classes should be autoloadable and the actual namespace of your controller classes match the value of this property. Otherwise, you will receive a "Page Not Found" error when accessing the application.

In case you want to break the convention as described above, you may configure the controllerMap property.

[[yii\base\Application::language|language]]

This property specifies the language in which the application should display content to end users. The default value of this property is `en`, meaning English. You should configure this property if your application needs to support multiple languages.

The value of this property determines various internationalization aspects, including message translation, date formatting, number formatting, etc. For example, the [[yii\jui\DatePicker]] widget will use this property value by default to determine in which language the calendar should be displayed and how the date should be formatted.

It is recommended that you specify a language in terms of an IETF language tag. For example, `en` stands for English, while `en-US` stands for English (United States).

More details about this property can be found in the Internationalization section.

[[yii\base\Application::modules|modules]]

This property specifies the modules that the application contains.

The property takes an array of module classes or configurations with the array keys being the module IDs. For example:

```
[
    'modules' => [
        // a "booking" module specified with the module class
        'booking' => 'app\modules\booking\BookingModule',

        // a "comment" module specified with a configuration array
        'comment' => [
            'class' => 'app\modules\comment\CommentModule',
            'db' => 'db',
        ],
    ],
]
```

Please refer to the Modules section for more details.

[[yii\base\Application::name|name]]

This property specifies the application name that may be displayed to end users. Unlike the [[yii\base\Application::id|id]] property, which should take a unique value, the value of this property is mainly for display purposes; it does not need to be unique.

You do not always need to configure this property if none of your code is using it.

[[yii\base\Application::params|params]]

This property specifies an array of globally accessible application parameters. Instead of using hardcoded numbers and strings everywhere in your code, it is a good practice to define them as application parameters in a single place and use the parameters in places where needed. For example, you may define the thumbnail image size as a parameter like the following:

```
[
    'params' => [
        'thumbnail.size' => [128, 128],
    ],
]
```

Then in your code where you need to use the size value, you can simply use code like the following:

```
$size = \Yii::$app->params['thumbnail.size'];
$width = \Yii::$app->params['thumbnail.size'][0];
```

Later if you decide to change the thumbnail size, you only need to modify it in the application configuration; you don't need to touch any dependent code.

[[yii\base\Application::sourceLanguage|sourceLanguage]]

This property specifies the language that the application code is written in. The default value is `'en-US'`, meaning English (United States). You should configure this property if the text content in your code is not in English.

Like the language property, you should configure this property in terms of an IETF language tag. For example, `en` stands for English, while `en-US` stands for English (United States).

More details about this property can be found in the Internationalization section.

[[yii\base\Application::timeZone|timeZone]]

This property is provided as an alternative way of setting the default time zone of the PHP runtime. By configuring this property, you are essentially calling the PHP function date_default_timezone_set(). For example:

```
[
    'timeZone' => 'America/Los_Angeles',
]
```

[[yii\base\Application::version|version]]

This property specifies the version of the application. It defaults to `'1.0'`. You do not need to configure this property if none of your code is using it.

Useful Properties

The properties described in this subsection are not commonly configured because their default values derive from common conventions. However, you may still configure them in case you want to break the conventions.

[[yii\base\Application::charset|charset]]

This property specifies the charset that the application uses. The default value is `'UTF-8'`, which should be kept as-is for most applications unless you are working with a legacy system that uses a lot of non-Unicode data.

[[yii\base\Application::defaultRoute|defaultRoute]]

This property specifies the route that an application should use when a request does not specify one. The route may consist of a child module ID, a controller ID, and/or an action ID. For example, `help`, `post/create`, or `admin/post/create`. If an action ID is not given, this property will take the default value specified in [[yii\base\Controller::defaultAction]].

For [[yii\web\Application|Web applications]], the default value of this property is `'site'`, which means the `SiteController` controller and its default action should be used. As a result, if you access the application without specifying a route, it will show the result of `app\controllers\SiteController::actionIndex()`.

For [[yii\console\Application|console applications]], the default value is `'help'`, which means the core command [[yii\console\controllers\HelpController::actionIndex()]] should be used. As a result, if you run the command `yii` without providing any arguments, it will display the help information.

[[yii\base\Application::extensions|extensions]]

This property specifies the list of extensions that are installed and used by the application. By default, it will take the array returned by the file `@vendor/yiisoft/extensions.php`. The `extensions.php` file is generated and maintained automatically when you use Composer to install extensions. So in most cases, you do not need to configure this property.

In the special case when you want to maintain extensions manually, you may configure this property as follows:

```
[
    'extensions' => [
        [
            'name' => 'extension name',
            'version' => 'version number',
            'bootstrap' => 'BootstrapClassName',  // optional, may also be a configura
tion array
            'alias' => [  // optional
                '@alias1' => 'to/path1',
                '@alias2' => 'to/path2',
            ],
        ],

        // ... more extensions like the above ...

    ],
]
```

As you can see, the property takes an array of extension specifications. Each extension is specified with an array consisting of `name` and `version` elements. If an extension needs to run during the bootstrap process, a `bootstrap` element may be specified with a bootstrapping class name or a configuration array. An extension may also define a few aliases.

[[yii\base\Application::layout|layout]]

This property specifies the name of the default layout that should be used when rendering a view. The default value is `'main'`, meaning the layout file `main.php` under the layout path should be used. If both of the layout path and the view path are taking the default values, the default layout file can be represented as the path alias `@app/views/layouts/main.php`.

You may configure this property to be `false` if you want to disable layout by default, although this is very rare.

[[yii\base\Application::layoutPath|layoutPath]]

This property specifies the path where layout files should be looked for. The default value is the `layouts` sub-directory under the view path. If the view path is taking its default value, the default layout path can be represented as the path alias `@app/views/layouts`.

You may configure it as a directory or a path alias.

[[yii\base\Application::runtimePath|runtimePath]]

This property specifies the path where temporary files, such as log files and cache files, can be generated. The default value is the directory represented by the alias `@app/runtime`.

You may configure it as a directory or a path alias. Note that the runtime path must be writable by the process running the application. And the path should be protected from being accessed by end users, because the temporary files under it may contain sensitive information.

To simplify access to this path, Yii has predefined a path alias named `@runtime` for it.

[[yii\base\Application::viewPath|viewPath]]

This property specifies the root directory where view files are located. The default value is the directory represented by the alias `@app/views`. You may configure it as a directory or a path alias.

[[yii\base\Application::vendorPath|vendorPath]]

This property specifies the vendor directory managed by Composer. It contains all third party libraries used by your application, including the Yii framework. The default value is the directory represented by the alias `@app/vendor`.

You may configure this property as a directory or a path alias. When you modify this property, make sure you also adjust the Composer configuration accordingly.

To simplify access to this path, Yii has predefined a path alias named `@vendor` for it.

[[yii\console\Application::enableCoreCommands|enableCoreCommands]]

This property is supported by [[yii\console\Application|console applications]] only. It specifies whether the core commands included in the Yii release should be enabled. The default value is `true`.

Application Events

An application triggers several events during the lifecycle of handling a request. You may attach event handlers to these events in application configurations as follows:

```
[
    'on beforeRequest' => function ($event) {
        // ...
    },
]
```

The use of the `on eventName` syntax is described in the Configurations section.

Alternatively, you may attach event handlers during the bootstrapping process after the application instance is created. For example:

```
\Yii::$app->on(\yii\base\Application::EVENT_BEFORE_REQUEST, function ($event) {
    // ...
});
```

[[yii\base\Application::EVENT_BEFORE_REQUEST|EVENT_BEFORE_REQUEST]]

This event is triggered *before* an application handles a request. The actual event name is `beforeRequest`.

When this event is triggered, the application instance has been configured and initialized. So it is a good place to insert your custom code via the event mechanism to intercept the request handling process. For example, in the event handler, you may dynamically set the [[yii\base\Application::language]] property based on some parameters.

[[yii\base\Application::EVENT_AFTER_REQUEST|EVENT_AFTER_REQUEST]]

This event is triggered *after* an application finishes handling a request but *before* sending the response. The actual event name is `afterRequest`.

When this event is triggered, the request handling is completed and you may take this chance to do some postprocessing of the request or customize the response.

Note that the [[yii\web\Response|response]] component also triggers some events while it is sending out response content to end users. Those events are triggered *after* this event.

[[yii\base\Application::EVENT_BEFORE_ACTION|EVENT_BEFORE_ACTION]]

This event is triggered *before* running every controller action. The actual event name is `beforeAction`.

The event parameter is an instance of [[yii\base\ActionEvent]]. An event handler may set the [[yii\base\ActionEvent::isValid]] property to be `false` to stop running the action. For example:

```
[
    'on beforeAction' => function ($event) {
        if (some condition) {
            $event->isValid = false;
        } else {
        }
    },
]
```

Note that the same `beforeAction` event is also triggered by modules and controllers. Application objects are the first ones triggering this event, followed by modules (if any), and finally controllers. If an event handler sets [[yii\base\ActionEvent::isValid]] to be `false`, all of the subsequent events will NOT be triggered.

[[yii\base\Application::EVENT_AFTER_ACTION|EVENT_AFTER_ACTION]]

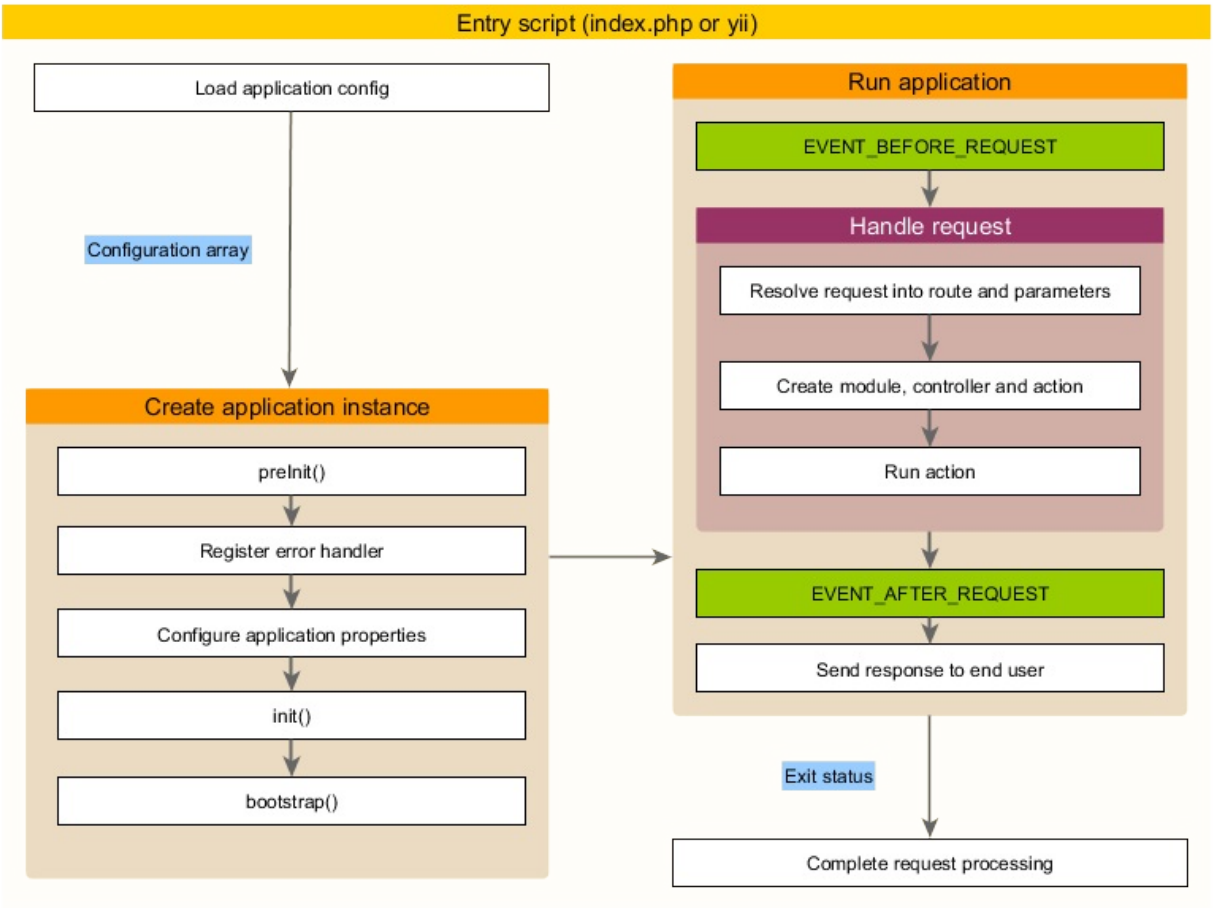
This event is triggered *after* running every controller action. The actual event name is `afterAction`.

The event parameter is an instance of [[yii\base\ActionEvent]]. Through the [[yii\base\ActionEvent::result]] property, an event handler may access or modify the action result. For example:

```
[
    'on afterAction' => function ($event) {
        if (some condition) {
            // modify $event->result
        } else {
        }
    },
]
```

Note that the same `afterAction` event is also triggered by modules and controllers. These objects trigger this event in the reverse order as for that of `beforeAction`. That is, controllers are the first objects triggering this event, followed by modules (if any), and finally applications.

Application Lifecycle



When an entry script is being executed to handle a request, an application will undergo the following lifecycle:

1. The entry script loads the application configuration as an array.
2. The entry script creates a new instance of the application:
    - [[yii\base\Application::preInit()|preInit()]] is called, which configures some high priority application properties, such as [[yii\base\Application::basePath|basePath]].
    - Register the [[yii\base\Application::errorHandler|error handler]].
    - Configure application properties.
    - [[yii\base\Application::init()|init()]] is called which further calls [[yii\base\Application::bootstrap()|bootstrap()]] to run bootstrapping components.
3. The entry script calls [[yii\base\Application::run()]] to run the application:
    - Trigger the [[yii\base\Application::EVENT_BEFORE_REQUEST|EVENT_BEFORE_REQUEST]] event.
    - Handle the request: resolve the request into a route and the associated parameters; create the module, controller, and action objects as specified by the route; and run the action.
    - Trigger the [[yii\base\Application::EVENT_AFTER_REQUEST|EVENT_AFTER_REQUEST]]

event.
    - Send response to the end user.
4. The entry script receives the exit status from the application and completes the request processing.

Application Components

Applications are service locators. They host a set of the so-called *application components* that provide different services for processing requests. For example, the `urlManager` component is responsible for routing Web requests to appropriate controllers; the `db` component provides DB-related services; and so on.

Each application component has an ID that uniquely identifies itself among other application components in the same application. You can access an application component through the expression:

```
\Yii::$app->componentID
```

For example, you can use `\Yii::$app->db` to get the [[yii\db\Connection|DB connection]], and `\Yii::$app->cache` to get the [[yii\caching\Cache|primary cache]] registered with the application.

An application component is created the first time it is accessed through the above expression. Any further accesses will return the same component instance.

Application components can be any objects. You can register them by configuring the [[yii\base\Application::components]] property in application configurations. For example,

```
[
    'components' => [
        // register "cache" component using a class name
        'cache' => 'yii\caching\ApcCache',

        // register "db" component using a configuration array
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],

        // register "search" component using an anonymous function
        'search' => function () {
            return new app\components\SolrService;
        },
    ],
]
```

> Info: While you can register as many application components as you want, you should do this judiciously. Application components are like global variables. Using too many application components can potentially make your code harder to test and maintain. In many cases, you can simply create a local component and use it when needed.

Bootstrapping Components

As mentioned above, an application component will only be instantiated when it is being accessed the first time. If it is not accessed at all during a request, it will not be instantiated. Sometimes, however, you may want to instantiate an application component for every request, even if it is not explicitly accessed. To do so, you may list its ID in the [[yii\base\Application::bootstrap|bootstrap]] property of the application.

For example, the following application configuration makes sure the `log` component is always loaded:

```
[
    'bootstrap' => [
        'log',
    ],
    'components' => [
        'log' => [
            // configuration for "log" component
        ],
    ],
]
```

Core Application Components

Yii defines a set of *core* application components with fixed IDs and default configurations. For example, the [[yii\web\Application::request|request]] component is used to collect information about a user request and resolve it into a route; the [[yii\base\Application::db|db]] component represents a database connection through which you can perform database queries. It is with help of these core application components that Yii applications are able to handle user requests.

Below is the list of the predefined core application components. You may configure and customize them like you do with normal application components. When you are configuring a core application component, if you do not specify its class, the default one will be used.

- [[yii\web\AssetManager|assetManager]]: manages asset bundles and asset publishing. Please refer to the Managing Assets section for more details.

- [[yii\db\Connection|db]]: represents a database connection through which you can perform DB queries. Note that when you configure this component, you must specify the component class as well as other required component properties, such as [[yii\db\Connection::dsn]]. Please refer to the Data Access Objects section for more details.
- [[yii\base\Application::errorHandler|errorHandler]]: handles PHP errors and exceptions. Please refer to the Handling Errors section for more details.
- [[yii\i18n\Formatter|formatter]]: formats data when they are displayed to end users. For example, a number may be displayed with thousand separator, a date may be formatted in long format. Please refer to the Data Formatting section for more details.
- [[yii\i18n\I18N|i18n]]: supports message translation and formatting. Please refer to the Internationalization section for more details.
- [[yii\log\Dispatcher|log]]: manages log targets. Please refer to the Logging section for more details.
- [[yii\swiftmailer\Mailer|mailer]]: supports mail composing and sending. Please refer to the Mailing section for more details.
- [[yii\base\Application::response|response]]: represents the response being sent to end users. Please refer to the Responses section for more details.
- [[yii\base\Application::request|request]]: represents the request received from end users. Please refer to the Requests section for more details.
- [[yii\web\Session|session]]: represents the session information. This component is only available in [[yii\web\Application|Web applications]]. Please refer to the Sessions and Cookies section for more details.
- [[yii\web\UrlManager|urlManager]]: supports URL parsing and creation. Please refer to the Routing and URL Creation section for more details.
- [[yii\web\User|user]]: represents the user authentication information. This component is only available in [[yii\web\Application|Web applications]]. Please refer to the Authentication section for more details.
- [[yii\web\View|view]]: supports view rendering. Please refer to the Views section for more details.

Controllers

Controllers are part of the MVC architecture. They are objects of classes extending from [[yii\base\Controller]] and are responsible for processing requests and generating responses. In particular, after taking over the control from applications, controllers will analyze incoming request data, pass them to models, inject model results into views, and finally generate outgoing responses.

Actions

Controllers are composed of *actions* which are the most basic units that end users can address and request for execution. A controller can have one or multiple actions.

The following example shows a `post` controller with two actions: `view` and `create`:

```php
namespace app\controllers;

use Yii;
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new NotFoundHttpException;
        }

        return $this->render('view', [
            'model' => $model,
        ]);
    }

    public function actionCreate()
    {
        $model = new Post;

        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('create', [
                'model' => $model,
            ]);
        }
    }
}
```

In the `view` action (defined by the `actionView()` method), the code first loads the model according to the requested model ID; If the model is loaded successfully, it will display it using a view named `view`. Otherwise, it will throw an exception.

In the `create` action (defined by the `actionCreate()` method), the code is similar. It first tries to populate a new instance of the model using the request data and save the model. If both succeed it will redirect the browser to the `view` action with the ID of the newly created model. Otherwise it will display the `create` view through which users can provide the needed input.

Routes

End users address actions through the so-called *routes*. A route is a string that consists of the following parts:

- a module ID: this exists only if the controller belongs to a non-application module;
- a controller ID: a string that uniquely identifies the controller among all controllers within the same application (or the same module if the controller belongs to a module);
- an action ID: a string that uniquely identifies the action among all actions within the same controller.

Routes take the following format:

```
ControllerID/ActionID
```

or the following format if the controller belongs to a module:

```
ModuleID/ControllerID/ActionID
```

So if a user requests with the URL `http://hostname/index.php?r=site/index`, the `index` action in the `site` controller will be executed. For more details on how routes are resolved into actions, please refer to the Routing and URL Creation section.

Creating Controllers

In [[yii\web\Application|Web applications]], controllers should extend from [[yii\web\Controller]] or its child classes. Similarly in [[yii\console\Application|console applications]], controllers should extend from [[yii\console\Controller]] or its child classes. The following code defines a `site` controller:

```php
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
}
```

Controller IDs

Usually, a controller is designed to handle the requests regarding a particular type of resource. For this reason, controller IDs are often nouns referring to the types of the resources that they are handling. For example, you may use `article` as the ID of a

controller that handles article data.

By default, controller IDs should contain these characters only: English letters in lower case, digits, underscores, hyphens, and forward slashes. For example, `article` and `post-comment` are both valid controller IDs, while `article?`, `PostComment`, `admin\post` are not.

A controller ID may also contain a subdirectory prefix. For example, `admin/article` stands for an `article` controller in the `admin` subdirectory under the [[yii\base\Application::controllerNamespace|controller namespace]]. Valid characters for subdirectory prefixes include: English letters in lower and upper cases, digits, underscores, and forward slashes, where forward slashes are used as separators for multi-level subdirectories (e.g. `panels/admin`).

Controller Class Naming

Controller class names can be derived from controller IDs according to the following procedure:

1. Turn the first letter in each word separated by hyphens into upper case. Note that if the controller ID contains slashes, this rule only applies to the part after the last slash in the ID.
2. Remove hyphens and replace any forward slashes with backward slashes.
3. Append the suffix `Controller`.
4. Prepend the [[yii\base\Application::controllerNamespace|controller namespace]].

The following are some examples, assuming the [[yii\base\Application::controllerNamespace|controller namespace]] takes the default value `app\controllers`:

- `article` becomes `app\controllers\ArticleController`;
- `post-comment` becomes `app\controllers\PostCommentController`;
- `admin/post-comment` becomes `app\controllers\admin\PostCommentController`;
- `adminPanels/post-comment` becomes `app\controllers\adminPanels\PostCommentController`.

Controller classes must be autoloadable. For this reason, in the above examples, the `article` controller class should be saved in the file whose alias is `@app/controllers/ArticleController.php`; while the `admin/post-comment` controller should be in `@app/controllers/admin/PostCommentController.php`.

> Info: The last example `admin/post-comment` shows how you can put a controller under a sub-directory of the [[yii\base\Application::controllerNamespace|controller namespace]]. This is useful when you want to organize your controllers into several categories and you do not want to use modules.

Controller Map

You can configure the [[yii\base\Application::controllerMap|controller map]] to overcome the constraints of the controller IDs and class names described above. This is mainly useful when you are using third-party controllers and you do not have control over their class names.

You may configure the [[yii\base\Application::controllerMap|controller map]] in the application configuration. For example:

```
[
    'controllerMap' => [
        // declares "account" controller using a class name
        'account' => 'app\controllers\UserController',

        // declares "article" controller using a configuration array
        'article' => [
            'class' => 'app\controllers\PostController',
            'enableCsrfValidation' => false,
        ],
    ],
]
```

Default Controller

Each application has a default controller specified via the [[yii\base\Application::defaultRoute]] property. When a request does not specify a route, the route specified by this property will be used. For [[yii\web\Application|Web applications]], its value is `'site'`, while for [[yii\console\Application|console applications]], it is `help`. Therefore, if a URL is `http://hostname/index.php`, then the `site` controller will handle the request.

You may change the default controller with the following application configuration:

```
[
    'defaultRoute' => 'main',
]
```

Creating Actions

Creating actions can be as simple as defining the so-called *action methods* in a controller class. An action method is a *public* method whose name starts with the word `action`. The return value of an action method represents the response data to be sent to end users. The

following code defines two actions, `index` and `hello-world`:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actionIndex()
    {
        return $this->render('index');
    }

    public function actionHelloWorld()
    {
        return 'Hello World';
    }
}
```

Action IDs

An action is often designed to perform a particular manipulation of a resource. For this reason, action IDs are usually verbs, such as `view`, `update`, etc.

By default, action IDs should contain these characters only: English letters in lower case, digits, underscores, and hyphens (you can use hyphens to separate words). For example, `view`, `update2`, and `comment-post` are all valid action IDs, while `view?` and `Update` are not.

You can create actions in two ways: inline actions and standalone actions. An inline action is defined as a method in the controller class, while a standalone action is a class extending [[yii\base\Action]] or its child classes. Inline actions take less effort to create and are often preferred if you have no intention to reuse these actions. Standalone actions, on the other hand, are mainly created to be used in different controllers or be redistributed as extensions.

Inline Actions

Inline actions refer to the actions that are defined in terms of action methods as we just described.

The names of the action methods are derived from action IDs according to the following procedure:

1. Turn the first letter in each word of the action ID into upper case.
2. Remove hyphens.
3. Prepend the prefix `action`.

For example, `index` becomes `actionIndex`, and `hello-world` becomes `actionHelloWorld`.

> Note: The names of the action methods are *case-sensitive*. If you have a method named `ActionIndex`, it will not be considered as an action method, and as a result, the request for the `index` action will result in an exception. Also note that action methods must be public. A private or protected method does NOT define an inline action.

Inline actions are the most commonly defined actions because they take little effort to create. However, if you plan to reuse the same action in different places, or if you want to redistribute an action, you should consider defining it as a *standalone action*.

Standalone Actions

Standalone actions are defined in terms of action classes extending [[yii\base\Action]] or its child classes. For example, in the Yii releases, there are [[yii\web\ViewAction]] and [[yii\web\ErrorAction]], both of which are standalone actions.

To use a standalone action, you should declare it in the *action map* by overriding the [[yii\base\Controller::actions()]] method in your controller classes like the following:

```
public function actions()
{
    return [
        // declares "error" action using a class name
        'error' => 'yii\web\ErrorAction',

        // declares "view" action using a configuration array
        'view' => [
            'class' => 'yii\web\ViewAction',
            'viewPrefix' => '',
        ],
    ];
}
```

As you can see, the `actions()` method should return an array whose keys are action IDs and values the corresponding action class names or configurations. Unlike inline actions, action IDs for standalone actions can contain arbitrary characters, as long as they are declared in the `actions()` method.

To create a standalone action class, you should extend [[yii\base\Action]] or a child class, and implement a public method named `run()`. The role of the `run()` method is similar to that of an action method. For example,

```php
<?php
namespace app\components;

use yii\base\Action;

class HelloWorldAction extends Action
{
    public function run()
    {
        return "Hello World";
    }
}
```

Action Results

The return value of an action method or of the `run()` method of a standalone action is significant. It stands for the result of the corresponding action.

The return value can be a response object which will be sent to the end user as the response.

- For [[yii\web\Application|Web applications]], the return value can also be some arbitrary data which will be assigned to [[yii\web\Response::data]] and be further converted into a string representing the response body.
- For [[yii\console\Application|console applications]], the return value can also be an integer representing the [[yii\console\Response::exitStatus|exit status]] of the command execution.

In the examples shown above, the action results are all strings which will be treated as the response body to be sent to end users. The following example shows how an action can redirect the user browser to a new URL by returning a response object (because the [[yii\web\Controller::redirect()|redirect()]] method returns a response object):

```php
public function actionForward()
{
    // redirect the user browser to http://example.com
    return $this->redirect('http://example.com');
}
```

Action Parameters

The action methods for inline actions and the `run()` methods for standalone actions can take parameters, called *action parameters*. Their values are obtained from requests. For [[yii\web\Application|Web applications]], the value of each action parameter is retrieved from

`$_GET` using the parameter name as the key; for [[yii\console\Application|console applications]], they correspond to the command line arguments.

In the following example, the `view` action (an inline action) has declared two parameters: `$id` and `$version`.

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public function actionView($id, $version = null)
    {
        // ...
    }
}
```

The action parameters will be populated as follows for different requests:

- `http://hostname/index.php?r=post/view&id=123`: the `$id` parameter will be filled with the value `'123'`, while `$version` is still `null` because there is no `version` query parameter.
- `http://hostname/index.php?r=post/view&id=123&version=2`: the `$id` and `$version` parameters will be filled with `'123'` and `'2'`, respectively.
- `http://hostname/index.php?r=post/view`: a [[yii\web\BadRequestHttpException]] exception will be thrown because the required `$id` parameter is not provided in the request.
- `http://hostname/index.php?r=post/view&id[]=123`: a [[yii\web\BadRequestHttpException]] exception will be thrown because `$id` parameter is receiving an unexpected array value `['123']`.

If you want an action parameter to accept array values, you should type-hint it with `array`, like the following:

```
public function actionView(array $id, $version = null)
{
    // ...
}
```

Now if the request is `http://hostname/index.php?r=post/view&id[]=123`, the `$id` parameter will take the value of `['123']`. If the request is `http://hostname/index.php?r=post/view&id=123`, the `$id` parameter will still receive the same array value because the scalar value `'123'` will be automatically turned into an array.

The above examples mainly show how action parameters work for Web applications. For console applications, please refer to the Console Commands section for more details.

Default Action

Each controller has a default action specified via the [[yii\base\Controller::defaultAction]] property. When a route contains the controller ID only, it implies that the default action of the specified controller is requested.

By default, the default action is set as `index`. If you want to change the default value, simply override this property in the controller class, like the following:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $defaultAction = 'home';

    public function actionHome()
    {
        return $this->render('home');
    }
}
```

Controller Lifecycle

When processing a request, an application will create a controller based on the requested route. The controller will then undergo the following lifecycle to fulfill the request:

1. The [[yii\base\Controller::init()]] method is called after the controller is created and configured.
2. The controller creates an action object based on the requested action ID:
   - If the action ID is not specified, the [[yii\base\Controller::defaultAction|default action ID]] will be used.
   - If the action ID is found in the [[yii\base\Controller::actions()|action map]], a standalone action will be created;
   - If the action ID is found to match an action method, an inline action will be created;
   - Otherwise an [[yii\base\InvalidRouteException]] exception will be thrown.
3. The controller sequentially calls the `beforeAction()` method of the application, the module (if the controller belongs to a module), and the controller.
   - If one of the calls returns `false`, the rest of the uncalled `beforeAction()` methods

will be skipped and the action execution will be cancelled.
    - By default, each `beforeAction()` method call will trigger a `beforeAction` event to which you can attach a handler.
4. The controller runs the action.
    - The action parameters will be analyzed and populated from the request data.
5. The controller sequentially calls the `afterAction()` method of the controller, the module (if the controller belongs to a module), and the application.
    - By default, each `afterAction()` method call will trigger an `afterAction` event to which you can attach a handler.
6. The application will take the action result and assign it to the response.

Best Practices

In a well-designed application, controllers are often very thin, with each action containing only a few lines of code. If your controller is rather complicated, it usually indicates that you should refactor it and move some code to other classes.

Here are some specific best practices. Controllers

- may access the request data;
- may call methods of models and other service components with request data;
- may use views to compose responses;
- should NOT process the request data - this should be done in the model layer;
- should avoid embedding HTML or other presentational code - this is better done in views.

Models

Models are part of the MVC architecture. They are objects representing business data, rules and logic.

You can create model classes by extending [[yii\base\Model]] or its child classes. The base class [[yii\base\Model]] supports many useful features:

- Attributes: represent the business data and can be accessed like normal object properties or array elements;
- Attribute labels: specify the display labels for attributes;
- Massive assignment: supports populating multiple attributes in a single step;
- Validation rules: ensures input data based on the declared validation rules;
- Data Exporting: allows model data to be exported in terms of arrays with customizable formats.

The `Model` class is also the base class for more advanced models, such as Active Record. Please refer to the relevant documentation for more details about these advanced models.

> Info: You are not required to base your model classes on [[yii\base\Model]]. However, because there are many Yii components built to support [[yii\base\Model]], it is usually the preferable base class for a model.

Attributes

Models represent business data in terms of *attributes*. Each attribute is like a publicly accessible property of a model. The method [[yii\base\Model::attributes()]] specifies what attributes a model class has.

You can access an attribute like accessing a normal object property:

```
$model = new \app\models\ContactForm;

// "name" is an attribute of ContactForm
$model->name = 'example';
echo $model->name;
```

You can also access attributes like accessing array elements, thanks to the support for ArrayAccess and Traversable by [[yii\base\Model]]:

```
$model = new \app\models\ContactForm;

// accessing attributes like array elements
$model['name'] = 'example';
echo $model['name'];

// Model is traversable using foreach.
foreach ($model as $name => $value) {
    echo "$name: $value\n";
}
```

Defining Attributes

By default, if your model class extends directly from [[yii\base\Model]], all its *non-static public* member variables are attributes. For example, the `ContactForm` model class below has four attributes: `name`, `email`, `subject` and `body`. The `ContactForm` model is used to represent the input data received from an HTML form.

```
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;
}
```

You may override [[yii\base\Model::attributes()]] to define attributes in a different way. The method should return the names of the attributes in a model. For example, [[yii\db\ActiveRecord]] does so by returning the column names of the associated database table as its attribute names. Note that you may also need to override the magic methods such as `__get()`, `__set()` so that the attributes can be accessed like normal object properties.

Attribute Labels

When displaying values or getting input for attributes, you often need to display some labels associated with attributes. For example, given an attribute named `firstName`, you may want to display a label `First Name` which is more user-friendly when displayed to end users in places such as form inputs and error messages.

You can get the label of an attribute by calling [[yii\base\Model::getAttributeLabel()]]. For example,

```
$model = new \app\models\ContactForm;

// displays "Name"
echo $model->getAttributeLabel('name');
```

By default, attribute labels are automatically generated from attribute names. The generation is done by the method [[yii\base\Model::generateAttributeLabel()]]. It will turn camel-case variable names into multiple words with the first letter in each word in upper case. For example, `username` becomes `Username`, and `firstName` becomes `First Name`.

If you do not want to use automatically generated labels, you may override [[yii\base\Model::attributeLabels()]] to explicitly declare attribute labels. For example,

```
namespace app\models;

use yii\base\Model;

class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;

    public function attributeLabels()
    {
        return [
            'name' => 'Your name',
            'email' => 'Your email address',
            'subject' => 'Subject',
            'body' => 'Content',
        ];
    }
}
```

For applications supporting multiple languages, you may want to translate attribute labels. This can be done in the [[yii\base\Model::attributeLabels()|attributeLabels()]] method as well, like the following:

```
public function attributeLabels()
{
    return [
        'name' => \Yii::t('app', 'Your name'),
        'email' => \Yii::t('app', 'Your email address'),
        'subject' => \Yii::t('app', 'Subject'),
        'body' => \Yii::t('app', 'Content'),
    ];
}
```

You may even conditionally define attribute labels. For example, based on the scenario the model is being used in, you may return different labels for the same attribute.

> Info: Strictly speaking, attribute labels are part of views. But declaring labels in models is often very convenient and can result in very clean and reusable code.

Scenarios

A model may be used in different *scenarios*. For example, a `User` model may be used to collect user login inputs, but it may also be used for the user registration purpose. In different scenarios, a model may use different business rules and logic. For example, the `email` attribute may be required during user registration, but not so during user login.

A model uses the [[yii\base\Model::scenario]] property to keep track of the scenario it is being used in. By default, a model supports only a single scenario named `default`. The following code shows two ways of setting the scenario of a model:

```
// scenario is set as a property
$model = new User;
$model->scenario = User::SCENARIO_LOGIN;

// scenario is set through configuration
$model = new User(['scenario' => User::SCENARIO_LOGIN]);
```

By default, the scenarios supported by a model are determined by the validation rules declared in the model. However, you can customize this behavior by overriding the [[yii\base\Model::scenarios()]] method, like the following:

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
        return [
            self::SCENARIO_LOGIN => ['username', 'password'],
            self::SCENARIO_REGISTER => ['username', 'email', 'password'],
        ];
    }
}
```

> Info: In the above and following examples, the model classes are extending from [[yii\db\ActiveRecord]] because the usage of multiple scenarios usually happens to Active Record classes.

The `scenarios()` method returns an array whose keys are the scenario names and values the corresponding *active attributes*. An active attribute can be massively assigned and is subject to validation. In the above example, the `username` and `password` attributes are active in the `login` scenario; while in the `register` scenario, `email` is also active besides `username` and `password`.

The default implementation of `scenarios()` will return all scenarios found in the validation rule declaration method [[yii\base\Model::rules()]]. When overriding `scenarios()`, if you want to introduce new scenarios in addition to the default ones, you may write code like the following:

```php
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    const SCENARIO_LOGIN = 'login';
    const SCENARIO_REGISTER = 'register';

    public function scenarios()
    {
        $scenarios = parent::scenarios();
        $scenarios[self::SCENARIO_LOGIN] = ['username', 'password'];
        $scenarios[self::SCENARIO_REGISTER] = ['username', 'email', 'password'];
        return $scenarios;
    }
}
```

The scenario feature is primarily used by validation and massive attribute assignment. You can, however, use it for other purposes. For example, you may declare attribute labels differently based on the current scenario.

Validation Rules

When the data for a model is received from end users, it should be validated to make sure it satisfies certain rules (called *validation rules*, also known as *business rules*). For example, given a `ContactForm` model, you may want to make sure all attributes are not empty and the `email` attribute contains a valid email address. If the values for some attributes do not satisfy the corresponding business rules, appropriate error messages should be displayed to help the user to fix the errors.

You may call [[yii\base\Model::validate()]] to validate the received data. The method will use the validation rules declared in [[yii\base\Model::rules()]] to validate every relevant attribute. If no error is found, it will return `true`. Otherwise, it will keep the errors in the [[yii\base\Model::errors]] property and return `false`. For example,

```
$model = new \app\models\ContactForm;

// populate model attributes with user inputs
$model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // all inputs are valid
} else {
    // validation failed: $errors is an array containing error messages
    $errors = $model->errors;
}
```

To declare validation rules associated with a model, override the [[yii\base\Model::rules()]] method by returning the rules that the model attributes should satisfy. The following example shows the validation rules declared for the `ContactForm` model:

```
public function rules()
{
    return [
        // the name, email, subject and body attributes are required
        [['name', 'email', 'subject', 'body'], 'required'],

        // the email attribute should be a valid email address
        ['email', 'email'],
    ];
}
```

A rule can be used to validate one or multiple attributes, and an attribute may be validated by one or multiple rules. Please refer to the Validating Input section for more details on how to declare validation rules.

Sometimes, you may want a rule to be applied only in certain scenarios. To do so, you can specify the `on` property of a rule, like the following:

```
public function rules()
{
    return [
        // username, email and password are all required in "register" scenario
        [['username', 'email', 'password'], 'required', 'on' => self::SCENARIO_REGISTE
R],

        // username and password are required in "login" scenario
        [['username', 'password'], 'required', 'on' => self::SCENARIO_LOGIN],
    ];
}
```

If you do not specify the `on` property, the rule would be applied in all scenarios. A rule is called an *active rule* if it can be applied in the current [[yii\base\Model::scenario|scenario]].

An attribute will be validated if and only if it is an active attribute declared in `scenarios()` and is associated with one or multiple active rules declared in `rules()`.

Massive Assignment

Massive assignment is a convenient way of populating a model with user inputs using a single line of code. It populates the attributes of a model by assigning the input data directly to the [[yii\base\Model::$attributes]] property. The following two pieces of code are equivalent, both trying to assign the form data submitted by end users to the attributes of the `ContactForm` model. Clearly, the former, which uses massive assignment, is much cleaner and less error prone than the latter:

```
$model = new \app\models\ContactForm;
$model->attributes = \Yii::$app->request->post('ContactForm');
```

```
$model = new \app\models\ContactForm;
$data = \Yii::$app->request->post('ContactForm', []);
$model->name = isset($data['name']) ? $data['name'] : null;
$model->email = isset($data['email']) ? $data['email'] : null;
$model->subject = isset($data['subject']) ? $data['subject'] : null;
$model->body = isset($data['body']) ? $data['body'] : null;
```

Safe Attributes

Massive assignment only applies to the so-called *safe attributes* which are the attributes listed in [[yii\base\Model::scenarios()]] for the current [[yii\base\Model::scenario|scenario]] of a model. For example, if the `User` model has the following scenario declaration, then when the current scenario is `login`, only the `username` and `password` can be massively assigned. Any other attributes will be kept untouched.

```
public function scenarios()
{
    return [
        self::SCENARIO_LOGIN => ['username', 'password'],
        self::SCENARIO_REGISTER => ['username', 'email', 'password'],
    ];
}
```

> Info: The reason that massive assignment only applies to safe attributes is because you want to control which attributes can be modified by end user data. For example, if the `User` model has a `permission` attribute which determines the permission assigned to the user, you would like this attribute to be modifiable by administrators through a backend interface only.

Because the default implementation of [[yii\base\Model::scenarios()]] will return all scenarios and attributes found in [[yii\base\Model::rules()]], if you do not override this method, it means an attribute is safe as long as it appears in one of the active validation rules.

For this reason, a special validator aliased `safe` is provided so that you can declare an attribute to be safe without actually validating it. For example, the following rules declare that both `title` and `description` are safe attributes.

```
public function rules()
{
    return [
        [['title', 'description'], 'safe'],
    ];
}
```

Unsafe Attributes

As described above, the [[yii\base\Model::scenarios()]] method serves for two purposes: determining which attributes should be validated, and determining which attributes are safe. In some rare cases, you may want to validate an attribute but do not want to mark it safe. You can do so by prefixing an exclamation mark `!` to the attribute name when declaring it in `scenarios()`, like the `secret` attribute in the following:

```
public function scenarios()
{
    return [
        self::SCENARIO_LOGIN => ['username', 'password', '!secret'],
    ];
}
```

When the model is in the `login` scenario, all three attributes will be validated. However, only the `username` and `password` attributes can be massively assigned. To assign an input value to the `secret` attribute, you have to do it explicitly as follows,

```
$model->secret = $secret;
```

The same can be done in `rules()` method:

```
public function rules()
{
    return [
        [['username', 'password', '!secret'], 'required', 'on' => 'login']
    ];
}
```

In this case attributes `username`, `password` and `secret` are required, but `secret` must be assigned explicitly.

Data Exporting

Models often need to be exported in different formats. For example, you may want to convert a collection of models into JSON or Excel format. The exporting process can be broken down into two independent steps:

- models are converted into arrays;
- the arrays are converted into target formats.

You may just focus on the first step, because the second step can be achieved by generic data formatters, such as [[yii\web\JsonResponseFormatter]].

The simplest way of converting a model into an array is to use the [[yii\base\Model::$attributes]] property. For example,

```
$post = \app\models\Post::findOne(100);
$array = $post->attributes;
```

By default, the [[yii\base\Model::$attributes]] property will return the values of *all* attributes declared in [[yii\base\Model::attributes()]].

A more flexible and powerful way of converting a model into an array is to use the [[yii\base\Model::toArray()]] method. Its default behavior is the same as that of [[yii\base\Model::$attributes]]. However, it allows you to choose which data items, called *fields*, to be put in the resulting array and how they should be formatted. In fact, it is the default way of exporting models in RESTful Web service development, as described in the Response Formatting.

Fields

A field is simply a named element in the array that is obtained by calling the [[yii\base\Model::toArray()]] method of a model.

By default, field names are equivalent to attribute names. However, you can change this behavior by overriding the [[yii\base\Model::fields()|fields()]] and/or [[yii\base\Model::extraFields()|extraFields()]] methods. Both methods should return a list of field definitions. The fields defined by `fields()` are default fields, meaning that `toArray()` will return these fields by default. The `extraFields()` method defines additionally available fields which can also be returned by `toArray()` as long as you specify them via the `$expand` parameter. For example, the following code will return all fields defined in `fields()` and the `prettyName` and `fullAddress` fields if they are defined in `extraFields()`.

```
$array = $model->toArray([], ['prettyName', 'fullAddress']);
```

You can override `fields()` to add, remove, rename or redefine fields. The return value of `fields()` should be an array. The array keys are the field names, and the array values are the corresponding field definitions which can be either property/attribute names or anonymous functions returning the corresponding field values. In the special case when a field name is the same as its defining attribute name, you can omit the array key. For example,

```
// explicitly list every field, best used when you want to make sure the changes
// in your DB table or model attributes do not cause your field changes (to keep API b
ackward compatibility).
public function fields()
{
    return [
        // field name is the same as the attribute name
        'id',

        // field name is "email", the corresponding attribute name is "email_address"
        'email' => 'email_address',

        // field name is "name", its value is defined by a PHP callback
        'name' => function () {
            return $this->first_name . ' ' . $this->last_name;
        },
    ];
}

// filter out some fields, best used when you want to inherit the parent implementation

// and blacklist some sensitive fields.
public function fields()
{
    $fields = parent::fields();

    // remove fields that contain sensitive information
    unset($fields['auth_key'], $fields['password_hash'], $fields['password_reset_token'
]);

    return $fields;
}
```

> Warning: Because by default all attributes of a model will be included in the exported array, you should examine your data to make sure they do not contain sensitive information. If there is such information, you should override `fields()` to filter them out. In the above example, we choose to filter out `auth_key`, `password_hash` and `password_reset_token`.

Best Practices

Models are the central places to represent business data, rules and logic. They often need to be reused in different places. In a well-designed application, models are usually much fatter than controllers.

In summary, models

- may contain attributes to represent business data;
- may contain validation rules to ensure the data validity and integrity;
- may contain methods implementing business logic;
- should NOT directly access request, session, or any other environmental data. These data should be injected by controllers into models;
- should avoid embedding HTML or other presentational code - this is better done in views;
- avoid having too many scenarios in a single model.

You may usually consider the last recommendation above when you are developing large complex systems. In these systems, models could be very fat because they are used in many places and may thus contain many sets of rules and business logic. This often ends up in a nightmare in maintaining the model code because a single touch of the code could affect several different places. To make the model code more maintainable, you may take the following strategy:

- Define a set of base model classes that are shared by different applications or modules. These model classes should contain minimal sets of rules and logic that are common among all their usages.
- In each application or module that uses a model, define a concrete model class by extending from the corresponding base model class. The concrete model classes should contain rules and logic that are specific for that application or module.

For example, in the Advanced Project Template, you may define a base model class `common\models\Post`. Then for the front end application, you define and use a concrete model class `frontend\models\Post` which extends from `common\models\Post`. And similarly for the back end application, you define `backend\models\Post`. With this strategy, you will be sure that the code in `frontend\models\Post` is only specific to the front end application, and if you make any change to it, you do not need to worry if the change may break the back end application.

Views

Views are part of the MVC architecture. They are code responsible for presenting data to end users. In a Web application, views are usually created in terms of *view templates* which are PHP script files containing mainly HTML code and presentational PHP code. They are managed by the [[yii\web\View|view]] application component which provides commonly used methods to facilitate view composition and rendering. For simplicity, we often call view templates or view template files as views.

Creating Views

As aforementioned, a view is simply a PHP script mixed with HTML and PHP code. The following is the view that presents a login form. As you can see, PHP code is used to generate the dynamic content, such as the page title and the form, while HTML code organizes them into a presentable HTML page.

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $form yii\widgets\ActiveForm */
/* @var $model app\models\LoginForm */

$this->title = 'Login';
?>
<h1><?= Html::encode($this->title) ?></h1>

<p>Please fill out the following fields to login:</p>

<?php $form = ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('Login') ?>
<?php ActiveForm::end(); ?>
```

Within a view, you can access `$this` which refers to the [[yii\web\View|view component]] managing and rendering this view template.

Besides `$this`, there may be other predefined variables in a view, such as `$model` in the above example. These variables represent the data that are *pushed* into the view by controllers or other objects which trigger the view rendering.

> Tip: The predefined variables are listed in a comment block at beginning of a view so that they can be recognized by IDEs. It is also a good way of documenting your views.

Security

When creating views that generate HTML pages, it is important that you encode and/or filter the data coming from end users before presenting them. Otherwise, your application may be subject to cross-site scripting attacks.

To display a plain text, encode it first by calling [[yii\helpers\Html::encode()]]. For example, the following code encodes the user name before displaying it:

```
<?php
use yii\helpers\Html;
?>

<div class="username">
    <?= Html::encode($user->name) ?>
</div>
```

To display HTML content, use [[yii\helpers\HtmlPurifier]] to filter the content first. For example, the following code filters the post content before displaying it:

```
<?php
use yii\helpers\HtmlPurifier;
?>

<div class="post">
    <?= HtmlPurifier::process($post->text) ?>
</div>
```

> Tip: While HTMLPurifier does excellent job in making output safe, it is not fast. You should consider caching the filtering result if your application requires high performance.

Organizing Views

Like controllers and models, there are conventions to organize views.

- For views rendered by a controller, they should be put under the directory `@app/views/ControllerID` by default, where `ControllerID` refers to the controller ID. For example, if the controller class is `PostController`, the directory would be `@app/views/post`; if it is `PostCommentController`, the directory would be `@app/views/post-comment`. In case the controller belongs to a module, the directory

would be `views/ControllerID` under the [[yii\base\Module::basePath|module directory]].
- For views rendered in a widget, they should be put under the `WidgetPath/views` directory by default, where `WidgetPath` stands for the directory containing the widget class file.
- For views rendered by other objects, it is recommended that you follow the similar convention as that for widgets.

You may customize these default view directories by overriding the [[yii\base\ViewContextInterface::getViewPath()]] method of controllers or widgets.

Rendering Views

You can render views in controllers, widgets, or any other places by calling view rendering methods. These methods share a similar signature shown as follows,

```
/**
 * @param string $view view name or file path, depending on the actual rendering metho
d
 * @param array $params the data to be passed to the view
 * @return string rendering result
 */
methodName($view, $params = [])
```

Rendering in Controllers

Within controllers, you may call the following controller methods to render views:

- [[yii\base\Controller::render()|render()]]: renders a named view and applies a layout to the rendering result.
- [[yii\base\Controller::renderPartial()|renderPartial()]]: renders a named view without any layout.
- [[yii\web\Controller::renderAjax()|renderAjax()]]: renders a named view without any layout, and injects all registered JS/CSS scripts and files. It is usually used in response to AJAX Web requests.
- [[yii\base\Controller::renderFile()|renderFile()]]: renders a view specified in terms of a view file path or alias.
- [[yii\base\Controller::renderContent()|renderContent()]]: renders a static string by embedding it into the currently applicable layout. This method is available since version 2.0.1.

For example,

```php
namespace app\controllers;

use Yii;
use app\models\Post;
use yii\web\Controller;
use yii\web\NotFoundHttpException;

class PostController extends Controller
{
    public function actionView($id)
    {
        $model = Post::findOne($id);
        if ($model === null) {
            throw new NotFoundHttpException;
        }

        // renders a view named "view" and applies a layout to it
        return $this->render('view', [
            'model' => $model,
        ]);
    }
}
```

Rendering in Widgets

Within widgets, you may call the following widget methods to render views.

- [[yii\base\Widget::render()|render()]]: renders a named view.
- [[yii\base\Widget::renderFile()|renderFile()]]: renders a view specified in terms of a view file path or alias.

For example,

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class ListWidget extends Widget
{
    public $items = [];

    public function run()
    {
        // renders a view named "list"
        return $this->render('list', [
            'items' => $this->items,
        ]);
    }
}
```

Rendering in Views

You can render a view within another view by calling one of the following methods provided by the [[yii\base\View|view component]]:

- [[yii\base\View::render()|render()]]: renders a named view.
- [[yii\web\View::renderAjax()|renderAjax()]]: renders a named view and injects all registered JS/CSS scripts and files. It is usually used in response to AJAX Web requests.
- [[yii\base\View::renderFile()|renderFile()]]: renders a view specified in terms of a view file path or alias.

For example, the following code in a view renders the `_overview.php` view file which is in the same directory as the view being currently rendered. Remember that `$this` in a view refers to the [[yii\base\View|view]] component:

```
<?= $this->render('_overview') ?>
```

Rendering in Other Places

In any place, you can get access to the [[yii\base\View|view]] application component by the expression `Yii::$app->view` and then call its aforementioned methods to render a view. For example,

```
// displays the view file "@app/views/site/license.php"
echo \Yii::$app->view->renderFile('@app/views/site/license.php');
```

Named Views

When you render a view, you can specify the view using either a view name or a view file path/alias. In most cases, you would use the former because it is more concise and flexible. We call views specified using names as *named views*.

A view name is resolved into the corresponding view file path according to the following rules:

- A view name may omit the file extension name. In this case, `.php` will be used as the extension. For example, the view name `about` corresponds to the file name `about.php`.
- If the view name starts with double slashes `//`, the corresponding view file path would be `@app/views/ViewName`. That is, the view is looked for under the [[yii\base\Application::viewPath|application's view path]]. For example, `//site/about` will be resolved into `@app/views/site/about.php`.
- If the view name starts with a single slash `/`, the view file path is formed by prefixing the view name with the [[yii\base\Module::viewPath|view path]] of the currently active module. If there is no active module, `@app/views/ViewName` will be used. For example, `/user/create` will be resolved into `@app/modules/user/views/user/create.php`, if the currently active module is `user`. If there is no active module, the view file path would be `@app/views/user/create.php`.
- If the view is rendered with a [[yii\base\View::context|context]] and the context implements [[yii\base\ViewContextInterface]], the view file path is formed by prefixing the [[yii\base\ViewContextInterface::getViewPath()|view path]] of the context to the view name. This mainly applies to the views rendered within controllers and widgets. For example, `about` will be resolved into `@app/views/site/about.php` if the context is the controller `SiteController`.
- If a view is rendered within another view, the directory containing the other view file will be prefixed to the new view name to form the actual view file path. For example, `item` will be resolved into `@app/views/post/item.php` if it is being rendered in the view `@app/views/post/index.php`.

According to the above rules, calling `$this->render('view')` in a controller `app\controllers\PostController` will actually render the view file `@app/views/post/view.php`, while calling `$this->render('_overview')` in that view will render the view file `@app/views/post/_overview.php`.

Accessing Data in Views

There are two approaches to access data within a view: push and pull.

By passing the data as the second parameter to the view rendering methods, you are using the push approach. The data should be represented as an array of name-value pairs. When the view is being rendered, the PHP `extract()` function will be called on this array so that the array is extracted into variables in the view. For example, the following view rendering code in a controller will push two variables to the `report` view: `$foo = 1` and `$bar = 2`.

```
echo $this->render('report', [
    'foo' => 1,
    'bar' => 2,
]);
```

The pull approach actively retrieves data from the [[yii\base\View|view component]] or other objects accessible in views (e.g. `Yii::$app`). Using the code below as an example, within the view you can get the controller object by the expression `$this->context`. And as a result, it is possible for you to access any properties or methods of the controller in the `report` view, such as the controller ID shown in the following:

```
The controller ID is: <?= $this->context->id ?>
```

The push approach is usually the preferred way of accessing data in views, because it makes views less dependent on context objects. Its drawback is that you need to manually build the data array all the time, which could become tedious and error prone if a view is shared and rendered in different places.

Sharing Data among Views

The [[yii\base\View|view component]] provides the [[yii\base\View::params|params]] property that you can use to share data among views.

For example, in an `about` view, you can have the following code which specifies the current segment of the breadcrumbs.

```
$this->params['breadcrumbs'][] = 'About Us';
```

Then, in the layout file, which is also a view, you can display the breadcrumbs using the data passed along [[yii\base\View::params|params]]:

```
<?= yii\widgets\Breadcrumbs::widget([
    'links' => isset($this->params['breadcrumbs']) ? $this->params['breadcrumbs'] : []
,
]) ?>
```

Layouts

Layouts are a special type of views that represent the common parts of multiple views. For example, the pages for most Web applications share the same page header and footer. While you can repeat the same page header and footer in every view, a better way is to do this once in a layout and embed the rendering result of a content view at an appropriate place in the layout.

Creating Layouts

Because layouts are also views, they can be created in the similar way as normal views. By default, layouts are stored in the directory `@app/views/layouts`. For layouts used within a module, they should be stored in the `views/layouts` directory under the [[yii\base\Module::basePath|module directory]]. You may customize the default layout directory by configuring the [[yii\base\Module::layoutPath]] property of the application or modules.

The following example shows how a layout looks like. Note that for illustrative purpose, we have greatly simplified the code in the layout. In practice, you may want to add more content to it, such as head tags, main menu, etc.

```
<?php
use yii\helpers\Html;

/* @var $this yii\web\View */
/* @var $content string */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <?= Html::csrfMetaTags() ?>
    <title><?= Html::encode($this->title) ?></title>
    <?php $this->head() ?>
</head>
<body>
<?php $this->beginBody() ?>
    <header>My Company</header>
    <?= $content ?>
    <footer>&copy; 2014 by My Company</footer>
<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

As you can see, the layout generates the HTML tags that are common to all pages. Within the `<body>` section, the layout echoes the `$content` variable which represents the rendering result of content views and is pushed into the layout when [[yii\base\Controller::render()]] is called.

Most layouts should call the following methods like shown in the above code. These methods mainly trigger events about the rendering process so that scripts and tags registered in other places can be properly injected into the places where these methods are called.

- [[yii\base\View::beginPage()|beginPage()]]: This method should be called at the very beginning of the layout. It triggers the [[yii\base\View::EVENT_BEGIN_PAGE|EVENT_BEGIN_PAGE]] event which indicates the beginning of a page.
- [[yii\base\View::endPage()|endPage()]]: This method should be called at the end of the layout. It triggers the [[yii\base\View::EVENT_END_PAGE|EVENT_END_PAGE]] event which indicates the end of a page.
- [[yii\web\View::head()|head()]]: This method should be called within the `<head>` section of an HTML page. It generates a placeholder which will be replaced with the registered head HTML code (e.g. link tags, meta tags) when a page finishes rendering.
- [[yii\web\View::beginBody()|beginBody()]]: This method should be called at the beginning of the `<body>` section. It triggers the [[yii\web\View::EVENT_BEGIN_BODY|EVENT_BEGIN_BODY]] event and generates a placeholder which will be replaced by the registered HTML code (e.g. JavaScript) targeted at the body begin position.
- [[yii\web\View::endBody()|endBody()]]: This method should be called at the end of the `<body>` section. It triggers the [[yii\web\View::EVENT_END_BODY|EVENT_END_BODY]] event and generates a placeholder which will be replaced by the registered HTML code (e.g. JavaScript) targeted at the body end position.

Accessing Data in Layouts

Within a layout, you have access to two predefined variables: `$this` and `$content`. The former refers to the [[yii\base\View|view]] component, like in normal views, while the latter contains the rendering result of a content view which is rendered by calling the [[yii\base\Controller::render()|render()]] method in controllers.

If you want to access other data in layouts, you have to use the pull method as described in the Accessing Data in Views subsection. If you want to pass data from a content view to a layout, you may use the method described in the Sharing Data among Views subsection.

Using Layouts

As described in the Rendering in Controllers subsection, when you render a view by calling the [[yii\base\Controller::render()|render()]] method in a controller, a layout will be applied to the rendering result. By default, the layout `@app/views/layouts/main.php` will be used.

You may use a different layout by configuring either [[yii\base\Application::layout]] or [[yii\base\Controller::layout]]. The former governs the layout used by all controllers, while the latter overrides the former for individual controllers. For example, the following code makes the `post` controller to use `@app/views/layouts/post.php` as the layout when rendering its views. Other controllers, assuming their `layout` property is untouched, will still use the default `@app/views/layouts/main.php` as the layout.

```
namespace app\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    public $layout = 'post';

    // ...
}
```

For controllers belonging to a module, you may also configure the module's [[yii\base\Module::layout|layout]] property to use a particular layout for these controllers.

Because the `layout` property may be configured at different levels (controllers, modules, application), behind the scene Yii takes two steps to determine what is the actual layout file being used for a particular controller.

In the first step, it determines the layout value and the context module:

- If the [[yii\base\Controller::layout]] property of the controller is not `null`, use it as the layout value and the [[yii\base\Controller::module|module]] of the controller as the context module.
- If the [[yii\base\Controller::layout]] property of the controller is `null`, search through all ancestor modules (including the application itself) of the controller and find the first module whose [[yii\base\Module::layout|layout]] property is not `null`. Use that module and its [[yii\base\Module::layout|layout]] value as the context module and the chosen layout value. If such a module cannot be found, it means no layout will be applied.

In the second step, it determines the actual layout file according to the layout value and the context module determined in the first step. The layout value can be:

- a path alias (e.g. `@app/views/layouts/main`).
- an absolute path (e.g. `/main`): the layout value starts with a slash. The actual layout file will be looked for under the application's [[yii\base\Application::layoutPath|layout path]] which defaults to `@app/views/layouts`.
- a relative path (e.g. `main`): the actual layout file will be looked for under the context module's [[yii\base\Module::layoutPath|layout path]] which defaults to the `views/layouts` directory under the [[yii\base\Module::basePath|module directory]].
- the boolean value `false`: no layout will be applied.

If the layout value does not contain a file extension, it will use the default one `.php`.

Nested Layouts

Sometimes you may want to nest one layout in another. For example, in different sections of a Web site, you want to use different layouts, while all these layouts share the same basic layout that generates the overall HTML5 page structure. You can achieve this goal by calling [[yii\base\View::beginContent()|beginContent()]] and [[yii\base\View::endContent()|endContent()]] in the child layouts like the following:

```
<?php $this->beginContent('@app/views/layouts/base.php'); ?>

...child layout content here...

<?php $this->endContent(); ?>
```

As shown above, the child layout content should be enclosed within [[yii\base\View::beginContent()|beginContent()]] and [[yii\base\View::endContent()|endContent()]]. The parameter passed to [[yii\base\View::beginContent()|beginContent()]] specifies what is the parent layout. It can be either a layout file or alias.

Using the above approach, you can nest layouts in more than one levels.

Using Blocks

Blocks allow you to specify the view content in one place while displaying it in another. They are often used together with layouts. For example, you can define a block in a content view and display it in the layout.

You call [[yii\base\View::beginBlock()|beginBlock()]] and [[yii\base\View::endBlock()|endBlock()]] to define a block. The block can then be accessed via `$view->blocks[$blockID]`, where `$blockID` stands for a unique ID that you assign to the block when defining it.

The following example shows how you can use blocks to customize specific parts of a layout in a content view.

First, in a content view, define one or multiple blocks:

```
...

<?php $this->beginBlock('block1'); ?>

...content of block1...

<?php $this->endBlock(); ?>

...

<?php $this->beginBlock('block3'); ?>

...content of block3...

<?php $this->endBlock(); ?>
```

Then, in the layout view, render the blocks if they are available, or display some default content if a block is not defined.

```
...
<?php if (isset($this->blocks['block1'])): ?>
    <?= $this->blocks['block1'] ?>
<?php else: ?>
    ... default content for block1 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block2'])): ?>
    <?= $this->blocks['block2'] ?>
<?php else: ?>
    ... default content for block2 ...
<?php endif; ?>

...

<?php if (isset($this->blocks['block3'])): ?>
    <?= $this->blocks['block3'] ?>
<?php else: ?>
    ... default content for block3 ...
<?php endif; ?>
...
```

Using View Components

[[yii\base\View|View components]] provides many view-related features. While you can get view components by creating individual instances of [[yii\base\View]] or its child class, in most cases you will mainly use the `view` application component. You can configure this component in application configurations like the following:

```
[
    // ...
    'components' => [
        'view' => [
            'class' => 'app\components\View',
        ],
        // ...
    ],
]
```

View components provide the following useful view-related features, each described in more details in a separate section:

- theming: allows you to develop and change the theme for your Web site.
- fragment caching: allows you to cache a fragment within a Web page.
- client script handling: supports CSS and JavaScript registration and rendering.
- asset bundle handling: supports registering and rendering of asset bundles.
- alternative template engines: allows you to use other template engines, such as Twig, Smarty.

You may also frequently use the following minor yet useful features when you are developing Web pages.

Setting Page Titles

Every Web page should have a title. Normally the title tag is being displayed in a layout. However, in practice the title is often determined in content views rather than layouts. To solve this problem, [[yii\web\View]] provides the [[yii\web\View::title|title]] property for you to pass the title information from content views to layouts.

To make use of this feature, in each content view, you can set the page title like the following:

```
<?php
$this->title = 'My page title';
?>
```

Then in the layout, make sure you have the following code in the `<head>` section:

```
<title><?= Html::encode($this->title) ?></title>
```

Registering Meta Tags

Web pages usually need to generate various meta tags needed by different parties. Like page titles, meta tags appear in the `<head>` section and are usually generated in layouts.

If you want to specify what meta tags to generate in content views, you can call [[yii\web\View::registerMetaTag()]] in a content view, like the following:

```
<?php
$this->registerMetaTag(['name' => 'keywords', 'content' => 'yii, framework, php']);
?>
```

The above code will register a "keywords" meta tag with the view component. The registered meta tag is rendered after the layout finishes rendering. The following HTML code will be generated and inserted at the place where you call [[yii\web\View::head()]] in the layout:

```
<meta name="keywords" content="yii, framework, php">
```

Note that if you call [[yii\web\View::registerMetaTag()]] multiple times, it will register multiple meta tags, regardless whether the meta tags are the same or not.

To make sure there is only a single instance of a meta tag type, you can specify a key as a second parameter when calling the method. For example, the following code registers two "description" meta tags. However, only the second one will be rendered.

```
$this->registerMetaTag(['name' => 'description', 'content' => 'This is my cool website
 made with Yii!'], 'description');
$this->registerMetaTag(['name' => 'description', 'content' => 'This website is about f
unny raccoons.'], 'description');
```

Registering Link Tags

Like meta tags, link tags are useful in many cases, such as customizing favicon, pointing to RSS feed or delegating OpenID to another server. You can work with link tags in the similar way as meta tags by using [[yii\web\View::registerLinkTag()]]. For example, in a content view, you can register a link tag like follows,

```
$this->registerLinkTag([
    'title' => 'Live News for Yii',
    'rel' => 'alternate',
    'type' => 'application/rss+xml',
    'href' => 'http://www.yiiframework.com/rss.xml/',
]);
```

The code above will result in

```
<link title="Live News for Yii" rel="alternate" type="application/rss+xml" href="http:
//www.yiiframework.com/rss.xml/">
```

Similar as [[yii\web\View::registerMetaTag()|registerMetaTag()]], you can specify a key when calling [[yii\web\View::registerLinkTag()|registerLinkTag()]] to avoid generating repeated link tags.

View Events

[[yii\base\View|View components]] trigger several events during the view rendering process. You may respond to these events to inject content into views or process the rendering results before they are sent to end users.

- [[yii\base\View::EVENT_BEFORE_RENDER|EVENT_BEFORE_RENDER]]: triggered at the beginning of rendering a file in a controller. Handlers of this event may set [[yii\base\ViewEvent::isValid]] to be `false` to cancel the rendering process.
- [[yii\base\View::EVENT_AFTER_RENDER|EVENT_AFTER_RENDER]]: triggered after rendering a file by the call of [[yii\base\View::afterRender()]]. Handlers of this event may obtain the rendering result through [[yii\base\ViewEvent::output]] and may modify this property to change the rendering result.
- [[yii\base\View::EVENT_BEGIN_PAGE|EVENT_BEGIN_PAGE]]: triggered by the call of [[yii\base\View::beginPage()]] in layouts.
- [[yii\base\View::EVENT_END_PAGE|EVENT_END_PAGE]]: triggered by the call of [[yii\base\View::endPage()]] in layouts.
- [[yii\web\View::EVENT_BEGIN_BODY|EVENT_BEGIN_BODY]]: triggered by the call of [[yii\web\View::beginBody()]] in layouts.
- [[yii\web\View::EVENT_END_BODY|EVENT_END_BODY]]: triggered by the call of [[yii\web\View::endBody()]] in layouts.

For example, the following code injects the current date at the end of the page body:

```
\Yii::$app->view->on(View::EVENT_END_BODY, function () {
    echo date('Y-m-d');
});
```

Rendering Static Pages

Static pages refer to those Web pages whose main content are mostly static without the need of accessing dynamic data pushed from controllers.

You can output static pages by putting their code in the view, and then using the code like the following in a controller:

```
public function actionAbout()
{
    return $this->render('about');
}
```

If a Web site contains many static pages, it would be very tedious repeating the similar code many times. To solve this problem, you may introduce a standalone action called [[yii\web\ViewAction]] in a controller. For example,

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'page' => [
                'class' => 'yii\web\ViewAction',
            ],
        ];
    }
}
```

Now if you create a view named `about` under the directory `@app/views/site/pages`, you will be able to display this view by the following URL:

```
http://localhost/index.php?r=site%2Fpage&view=about
```

The `GET` parameter `view` tells [[yii\web\ViewAction]] which view is requested. The action will then look for this view under the directory `@app/views/site/pages`. You may configure [[yii\web\ViewAction::viewPrefix]] to change the directory for searching these views.

Best Practices

Views are responsible for presenting models in the format that end users desire. In general, views

- should mainly contain presentational code, such as HTML, and simple PHP code to traverse, format and render data.
- should not contain code that performs DB queries. Such code should be done in models.
- should avoid direct access to request data, such as `$_GET`, `$_POST`. This belongs to controllers. If request data is needed, they should be pushed into views by controllers.
- may read model properties, but should not modify them.

To make views more manageable, avoid creating views that are too complex or contain too much redundant code. You may use the following techniques to achieve this goal:

- use layouts to represent common presentational sections (e.g. page header, footer).
- divide a complicated view into several smaller ones. The smaller views can be rendered and assembled into a bigger one using the rendering methods that we have described.
- create and use widgets as building blocks of views.
- create and use helper classes to transform and format data in views.

Modules

Modules are self-contained software units that consist of models, views, controllers, and other supporting components. End users can access the controllers of a module when it is installed in application. For these reasons, modules are often viewed as mini-applications. Modules differ from applications in that modules cannot be deployed alone and must reside within applications.

Creating Modules

A module is organized as a directory which is called the [[yii\base\Module::basePath|base path]] of the module. Within the directory, there are sub-directories, such as `controllers`, `models`, `views`, which hold controllers, models, views, and other code, just like in an application. The following example shows the content within a module:

```
forum/
    Module.php                  the module class file
    controllers/                containing controller class files
        DefaultController.php   the default controller class file
    models/                     containing model class files
    views/                      containing controller view and layout files
        layouts/                containing layout view files
        default/                containing view files for DefaultController
            index.php           the index view file
```

Module Classes

Each module should have a unique module class which extends from [[yii\base\Module]]. The class should be located directly under the module's [[yii\base\Module::basePath|base path]] and should be autoloadable. When a module is being accessed, a single instance of the corresponding module class will be created. Like application instances, module instances are used to share data and components for code within modules.

The following is an example how a module class may look like:

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->params['foo'] = 'bar';
        // ...  other initialization code ...
    }
}
```

If the `init()` method contains a lot of code initializing the module's properties, you may also save them in terms of a configuration and load it with the following code in `init()`:

```
public function init()
{
    parent::init();
    // initialize the module with the configuration loaded from config.php
    \Yii::configure($this, require(__DIR__ . '/config.php'));
}
```

where the configuration file `config.php` may contain the following content, similar to that in an application configuration.

```
<?php
return [
    'components' => [
        // list of component configurations
    ],
    'params' => [
        // list of parameters
    ],
];
```

Controllers in Modules

When creating controllers in a module, a convention is to put the controller classes under the `controllers` sub-namespace of the namespace of the module class. This also means the controller class files should be put in the `controllers` directory within the module's [[yii\base\Module::basePath|base path]]. For example, to create a `post` controller in the `forum` module shown in the last subsection, you should declare the controller class like the following:

```
namespace app\modules\forum\controllers;

use yii\web\Controller;

class PostController extends Controller
{
    // ...
}
```

You may customize the namespace of controller classes by configuring the [[yii\base\Module::controllerNamespace]] property. In case some of the controllers are outside of this namespace, you may make them accessible by configuring the [[yii\base\Module::controllerMap]] property, similar to what you do in an application.

Views in Modules

Views in a module should be put in the `views` directory within the module's [[yii\base\Module::basePath|base path]]. For views rendered by a controller in the module, they should be put under the directory `views/ControllerID`, where `ControllerID` refers to the controller ID. For example, if the controller class is `PostController`, the directory would be `views/post` within the module's [[yii\base\Module::basePath|base path]].

A module can specify a layout that is applied to the views rendered by the module's controllers. The layout should be put in the `views/layouts` directory by default, and you should configure the [[yii\base\Module::layout]] property to point to the layout name. If you do not configure the `layout` property, the application's layout will be used instead.

Console commands in Modules

Your module may also declare commands, that will be available through the Console mode.

In order for the command line utility to see your commands, you will need to change the [[yii\base\Module::controllerNamespace]] property, when Yii is executed in the console mode, and point it to your commands namespace.

One way to achieve that is to test the instance type of the Yii application in the module's `init()` method:

```
public function init()
{
    parent::init();
    if (Yii::$app instanceof \yii\console\Application) {
        $this->controllerNamespace = 'app\modules\forum\commands';
    }
}
```

Your commands will then be available from the command line using the following route:

```
yii <module_id>/<command>/<sub_command>
```

Using Modules

To use a module in an application, simply configure the application by listing the module in the [[yii\base\Application::modules|modules]] property of the application. The following code in the application configuration uses the `forum` module:

```
[
    'modules' => [
        'forum' => [
            'class' => 'app\modules\forum\Module',
            // ... other configurations for the module ...
        ],
    ],
]
```

The [[yii\base\Application::modules|modules]] property takes an array of module configurations. Each array key represents a *module ID* which uniquely identifies the module among all modules in the application, and the corresponding array value is a configuration for creating the module.

Routes

Like accessing controllers in an application, routes are used to address controllers in a module. A route for a controller within a module must begin with the module ID followed by the controller ID and action ID. For example, if an application uses a module named `forum`, then the route `forum/post/index` would represent the `index` action of the `post` controller in the module. If the route only contains the module ID, then the [[yii\base\Module::defaultRoute]] property, which defaults to `default`, will determine which controller/action should be used. This means a route `forum` would represent the `default` controller in the `forum` module.

Accessing Modules

Within a module, you may often need to get the instance of the module class so that you can access the module ID, module parameters, module components, etc. You can do so by using the following statement:

```
$module = MyModuleClass::getInstance();
```

where `MyModuleClass` refers to the name of the module class that you are interested in. The `getInstance()` method will return the currently requested instance of the module class. If the module is not requested, the method will return `null`. Note that you do not want to manually create a new instance of the module class because it will be different from the one created by Yii in response to a request.

> Info: When developing a module, you should not assume the module will use a fixed ID. This is because a module can be associated with an arbitrary ID when used in an application or within another module. In order to get the module ID, you should use the above approach to get the module instance first, and then get the ID via `$module->id`.

You may also access the instance of a module using the following approaches:

```
// get the child module whose ID is "forum"
$module = \Yii::$app->getModule('forum');

// get the module to which the currently requested controller belongs
$module = \Yii::$app->controller->module;
```

The first approach is only useful when you know the module ID, while the second approach is best used when you know about the controllers being requested.

Once you have the module instance, you can access parameters and components registered with the module. For example,

```
$maxPostCount = $module->params['maxPostCount'];
```

Bootstrapping Modules

Some modules may need to be run for every request. The [[yii\debug\Module|debug]] module is such an example. To do so, list the IDs of such modules in the [[yii\base\Application::bootstrap|bootstrap]] property of the application.

For example, the following application configuration makes sure the `debug` module is always loaded:

```
[
    'bootstrap' => [
        'debug',
    ],

    'modules' => [
        'debug' => 'yii\debug\Module',
    ],
]
```

Nested Modules

Modules can be nested in unlimited levels. That is, a module can contain another module which can contain yet another module. We call the former *parent module* while the latter *child module*. Child modules must be declared in the [[yii\base\Module::modules|modules]] property of their parent modules. For example,

```
namespace app\modules\forum;

class Module extends \yii\base\Module
{
    public function init()
    {
        parent::init();

        $this->modules = [
            'admin' => [
                // you should consider using a shorter namespace here!
                'class' => 'app\modules\forum\modules\admin\Module',
            ],
        ];
    }
}
```

For a controller within a nested module, its route should include the IDs of all its ancestor modules. For example, the route `forum/admin/dashboard/index` represents the `index` action of the `dashboard` controller in the `admin` module which is a child module of the `forum` module.

> Info: The [[yii\base\Module::getModule()|getModule()]] method only returns the child module directly belonging to its parent. The [[yii\base\Application::loadedModules]] property keeps a list of loaded modules, including both direct children and nested ones, indexed by their class names.

Best Practices

Modules are best used in large applications whose features can be divided into several groups, each consisting of a set of closely related features. Each such feature group can be developed as a module which is developed and maintained by a specific developer or team.

Modules are also a good way of reusing code at the feature group level. Some commonly used features, such as user management, comment management, can all be developed in terms of modules so that they can be reused easily in future projects.

Filters

Filters are objects that run before and/or after controller actions. For example, an access control filter may run before actions to ensure that they are allowed to be accessed by particular end users; a content compression filter may run after actions to compress the response content before sending them out to end users.

A filter may consist of a pre-filter (filtering logic applied *before* actions) and/or a post-filter (logic applied *after* actions).

Using Filters

Filters are essentially a special kind of behaviors. Therefore, using filters is the same as using behaviors. You can declare filters in a controller class by overriding its [[yii\base\Controller::behaviors()|behaviors()]] method like the following:

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index', 'view'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

By default, filters declared in a controller class will be applied to *all* actions in that controller. You can, however, explicitly specify which actions the filter should be applied to by configuring the [[yii\base\ActionFilter::only|only]] property. In the above example, the `HttpCache` filter only applies to the `index` and `view` actions. You can also configure the [[yii\base\ActionFilter::except|except]] property to blacklist some actions from being filtered.

Besides controllers, you can also declare filters in a module or application. When you do so, the filters will be applied to *all* controller actions belonging to that module or application, unless you configure the filters' [[yii\base\ActionFilter::only|only]] and [[yii\base\ActionFilter::except|except]] properties like described above.

> Note: When declaring filters in modules or applications, you should use routes instead of action IDs in the [[yii\base\ActionFilter::only|only]] and [[yii\base\ActionFilter::except|except]] properties. This is because action IDs alone cannot fully specify actions within the scope of a module or application.

When multiple filters are configured for a single action, they are applied according to the rules described below:

- Pre-filtering
  - Apply filters declared in the application in the order they are listed in `behaviors()`.
  - Apply filters declared in the module in the order they are listed in `behaviors()`.
  - Apply filters declared in the controller in the order they are listed in `behaviors()`.
  - If any of the filters cancel the action execution, the filters (both pre-filters and post-filters) after it will not be applied.
- Running the action if it passes the pre-filtering.
- Post-filtering
  - Apply filters declared in the controller in the reverse order they are listed in `behaviors()`.
  - Apply filters declared in the module in the reverse order they are listed in `behaviors()`.
  - Apply filters declared in the application in the reverse order they are listed in `behaviors()`.

Creating Filters

To create a new action filter, extend from [[yii\base\ActionFilter]] and override the [[yii\base\ActionFilter::beforeAction()|beforeAction()]] and/or [[yii\base\ActionFilter::afterAction()|afterAction()]] methods. The former will be executed before an action runs while the latter after an action runs. The return value of [[yii\base\ActionFilter::beforeAction()|beforeAction()]] determines whether an action should be executed or not. If it is `false`, the filters after this one will be skipped and the action will not be executed.

The following example shows a filter that logs the action execution time:

```php
namespace app\components;

use Yii;
use yii\base\ActionFilter;

class ActionTimeFilter extends ActionFilter
{
    private $_startTime;

    public function beforeAction($action)
    {
        $this->_startTime = microtime(true);
        return parent::beforeAction($action);
    }

    public function afterAction($action, $result)
    {
        $time = microtime(true) - $this->_startTime;
        Yii::trace("Action '{$action->uniqueId}' spent $time second.");
        return parent::afterAction($action, $result);
    }
}
```

Core Filters

Yii provides a set of commonly used filters, found primarily under the `yii\filters` namespace. In the following, we will briefly introduce these filters.

[[yii\filters\AccessControl|AccessControl]]

AccessControl provides simple access control based on a set of [[yii\filters\AccessControl::rules|rules]]. In particular, before an action is executed, AccessControl will examine the listed rules and find the first one that matches the current context variables (such as user IP address, user login status, etc.) The matching rule will dictate whether to allow or deny the execution of the requested action. If no rule matches, the access will be denied.

The following example shows how to allow authenticated users to access the `create` and `update` actions while denying all other users from accessing these two actions.

```
use yii\filters\AccessControl;

public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'only' => ['create', 'update'],
            'rules' => [
                // allow authenticated users
                [
                    'allow' => true,
                    'roles' => ['@'],
                ],
                // everything else is denied by default
            ],
        ],
    ];
}
```

For more details about access control in general, please refer to the Authorization section.

Authentication Method Filters

Authentication method filters are used to authenticate a user using various methods, such as HTTP Basic Auth, OAuth 2. These filter classes are all under the `yii\filters\auth` namespace.

The following example shows how you can use [[yii\filters\auth\HttpBasicAuth]] to authenticate a user using an access token based on HTTP Basic Auth method. Note that in order for this to work, your [[yii\web\User::identityClass|user identity class]] must implement the [[yii\web\IdentityInterface::findIdentityByAccessToken()|findIdentityByAccessToken()]] method.

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    return [
        'basicAuth' => [
            'class' => HttpBasicAuth::className(),
        ],
    ];
}
```

Authentication method filters are commonly used in implementing RESTful APIs. For more details, please refer to the RESTful Authentication section.

[[yii\filters\ContentNegotiator|ContentNegotiator]]

ContentNegotiator supports response format negotiation and application language negotiation. It will try to determine the response format and/or language by examining `GET` parameters and `Accept` HTTP header.

In the following example, ContentNegotiator is configured to support JSON and XML response formats, and English (United States) and German languages.

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

public function behaviors()
{
    return [
        [
            'class' => ContentNegotiator::className(),
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ];
}
```

Response formats and languages often need to be determined much earlier during the application lifecycle. For this reason, ContentNegotiator is designed in a way such that it can also be used as a bootstrapping component besides being used as a filter. For example, you may configure it in the application configuration like the following:

```
use yii\filters\ContentNegotiator;
use yii\web\Response;

[
    'bootstrap' => [
        [
            'class' => ContentNegotiator::className(),
            'formats' => [
                'application/json' => Response::FORMAT_JSON,
                'application/xml' => Response::FORMAT_XML,
            ],
            'languages' => [
                'en-US',
                'de',
            ],
        ],
    ],
];
```

> Info: In case the preferred content type and language cannot be determined from a request, the first format and language listed in [[formats]] and [[languages]] will be used.

[[yii\filters\HttpCache|HttpCache]]

HttpCache implements client-side caching by utilizing the `Last-Modified` and `Etag` HTTP headers. For example,

```
use yii\filters\HttpCache;

public function behaviors()
{
    return [
        [
            'class' => HttpCache::className(),
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

Please refer to the HTTP Caching section for more details about using HttpCache.

[[yii\filters\PageCache|PageCache]]

PageCache implements server-side caching of whole pages. In the following example, PageCache is applied to the `index` action to cache the whole page for maximum 60 seconds or until the count of entries in the `post` table changes. It also stores different versions of the page depending on the chosen application language.

```
use yii\filters\PageCache;
use yii\caching\DbDependency;

public function behaviors()
{
    return [
        'pageCache' => [
            'class' => PageCache::className(),
            'only' => ['index'],
            'duration' => 60,
            'dependency' => [
                'class' => DbDependency::className(),
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
            'variations' => [
                \Yii::$app->language,
            ]
        ],
    ];
}
```

Please refer to the Page Caching section for more details about using PageCache.

[[yii\filters\RateLimiter|RateLimiter]]

RateLimiter implements a rate limiting algorithm based on the leaky bucket algorithm. It is primarily used in implementing RESTful APIs. Please refer to the Rate Limiting section for details about using this filter.

[[yii\filters\VerbFilter|VerbFilter]]

VerbFilter checks if the HTTP request methods are allowed by the requested actions. If not allowed, it will throw an HTTP 405 exception. In the following example, VerbFilter is declared to specify a typical set of allowed request methods for CRUD actions.

```
use yii\filters\VerbFilter;

public function behaviors()
{
    return [
        'verbs' => [
            'class' => VerbFilter::className(),
            'actions' => [
                'index'  => ['get'],
                'view'   => ['get'],
                'create' => ['get', 'post'],
                'update' => ['get', 'put', 'post'],
                'delete' => ['post', 'delete'],
            ],
        ],
    ];
}
```

[[yii\filters\Cors|Cors]]

Cross-origin resource sharing CORS is a mechanism that allows many resources (e.g. fonts, JavaScript, etc.) on a Web page to be requested from another domain outside the domain the resource originated from. In particular, JavaScript's AJAX calls can use the XMLHttpRequest mechanism. Such "cross-domain" requests would otherwise be forbidden by Web browsers, per the same origin security policy. CORS defines a way in which the browser and the server can interact to determine whether or not to allow the cross-origin request.

The [[yii\filters\Cors|Cors filter]] should be defined before Authentication / Authorization filters to make sure the CORS headers will always be sent.

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),
        ],
    ], parent::behaviors());
}
```

Also check the section on REST Controllers if you want to add the CORS filter to an [[yii\rest\ActiveController]] class in your API.

The Cors filtering could be tuned using the [[yii\filters\Cors::$cors|$cors]] property.

- `cors['Origin']` : array used to define allowed origins. Can be `['*']` (everyone) or `['http://www.myserver.net', 'http://www.myotherserver.com']` . Default to `['*']` .
- `cors['Access-Control-Request-Method']` : array of allowed verbs like `['GET', 'OPTIONS', 'HEAD']` . Default to `['GET', 'POST', 'PUT', 'PATCH', 'DELETE', 'HEAD', 'OPTIONS']` .
- `cors['Access-Control-Request-Headers']` : array of allowed headers. Can be `['*']` all headers or specific ones `['X-Request-With']` . Default to `['*']` .
- `cors['Access-Control-Allow-Credentials']` : define if current request can be made using credentials. Can be `true` , `false` or `null` (not set). Default to `null` .
- `cors['Access-Control-Max-Age']` : define lifetime of pre-flight request. Default to `86400` .

For example, allowing CORS for origin : `http://www.myserver.net` with method `GET` , `HEAD` and `OPTIONS` :

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),
            'cors' => [
                'Origin' => ['http://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD', 'OPTIONS'],
            ],
        ],
    ], parent::behaviors());
}
```

You may tune the CORS headers by overriding default parameters on a per action basis. For example adding the `Access-Control-Allow-Credentials` for the `login` action could be done like this :

```
use yii\filters\Cors;
use yii\helpers\ArrayHelper;

public function behaviors()
{
    return ArrayHelper::merge([
        [
            'class' => Cors::className(),
            'cors' => [
                'Origin' => ['http://www.myserver.net'],
                'Access-Control-Request-Method' => ['GET', 'HEAD', 'OPTIONS'],
            ],
            'actions' => [
                'login' => [
                    'Access-Control-Allow-Credentials' => true,
                ]
            ]
        ],
    ], parent::behaviors());
}
```

Widgets

Widgets are reusable building blocks used in views to create complex and configurable user interface elements in an object-oriented fashion. For example, a date picker widget may generate a fancy date picker that allows users to pick a date as their input. All you need to do is just to insert the code in a view like the following:

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget(['name' => 'date']) ?>
```

There are a good number of widgets bundled with Yii, such as [[yii\widgets\ActiveForm|active form]], [[yii\widgets\Menu|menu]], jQuery UI widgets, Twitter Bootstrap widgets. In the following, we will introduce the basic knowledge about widgets. Please refer to the class API documentation if you want to learn about the usage of a particular widget.

Using Widgets

Widgets are primarily used in views. You can call the [[yii\base\Widget::widget()]] method to use a widget in a view. The method takes a configuration array for initializing the widget and returns the rendering result of the widget. For example, the following code inserts a date picker widget which is configured to use the Russian language and keep the input in the `from_date` attribute of `$model`.

```
<?php
use yii\jui\DatePicker;
?>
<?= DatePicker::widget([
    'model' => $model,
    'attribute' => 'from_date',
    'language' => 'ru',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]) ?>
```

Some widgets can take a block of content which should be enclosed between the invocation of [[yii\base\Widget::begin()]] and [[yii\base\Widget::end()]]. For example, the following code uses the [[yii\widgets\ActiveForm]] widget to generate a login form. The widget will generate the opening and closing `<form>` tags at the place where `begin()` and `end()` are called, respectively. Anything in between will be rendered as is.

```
<?php
use yii\widgets\ActiveForm;
use yii\helpers\Html;
?>

<?php $form = ActiveForm::begin(['id' => 'login-form']); ?>

    <?= $form->field($model, 'username') ?>

    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <?= Html::submitButton('Login') ?>
    </div>

<?php ActiveForm::end(); ?>
```

Note that unlike [[yii\base\Widget::widget()]] which returns the rendering result of a widget, the method [[yii\base\Widget::begin()]] returns an instance of the widget which you can use to build the widget content.

> Note: Some widgets will use output buffering to adjust the enclosed content when [[yii\base\Widget::end()]] is called. For this reason calling [[yii\base\Widget::begin()]] and [[yii\base\Widget::end()]] is expected to happen in the same view file. Not following this rule may result in unexpected output.

Configuring global defaults

Global defaults for a widget type could be configured via DI container:

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

See "Practical Usage" section in Dependency Injection Container guide for details.

Creating Widgets

To create a widget, extend from [[yii\base\Widget]] and override the [[yii\base\Widget::init()]] and/or [[yii\base\Widget::run()]] methods. Usually, the `init()` method should contain the code that normalizes the widget properties, while the `run()` method should contain the code that generates the rendering result of the widget. The rendering result may be directly "echoed" or returned as a string by `run()`.

In the following example, `HelloWidget` HTML-encodes and displays the content assigned to its `message` property. If the property is not set, it will display "Hello World" by default.

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public $message;

    public function init()
    {
        parent::init();
        if ($this->message === null) {
            $this->message = 'Hello World';
        }
    }

    public function run()
    {
        return Html::encode($this->message);
    }
}
```

To use this widget, simply insert the following code in a view:

```
<?php
use app\components\HelloWidget;
?>
<?= HelloWidget::widget(['message' => 'Good morning']) ?>
```

Below is a variant of `HelloWidget` which takes the content enclosed within the `begin()` and `end()` calls, HTML-encodes it and then displays it.

```
namespace app\components;

use yii\base\Widget;
use yii\helpers\Html;

class HelloWidget extends Widget
{
    public function init()
    {
        parent::init();
        ob_start();
    }

    public function run()
    {
        $content = ob_get_clean();
        return Html::encode($content);
    }
}
```

As you can see, PHP's output buffer is started in `init()` so that any output between the calls of `init()` and `run()` can be captured, processed and returned in `run()`.

> Info: When you call [[yii\base\Widget::begin()]], a new instance of the widget will be created and the `init()` method will be called at the end of the widget constructor. When you call [[yii\base\Widget::end()]], the `run()` method will be called whose return result will be echoed by `end()`.

The following code shows how to use this new variant of `HelloWidget`:

```
<?php
use app\components\HelloWidget;
?>
<?php HelloWidget::begin(); ?>

    content that may contain <tag>'s

<?php HelloWidget::end(); ?>
```

Sometimes, a widget may need to render a big chunk of content. While you can embed the content within the `run()` method, a better approach is to put it in a view and call [[yii\base\Widget::render()]] to render it. For example,

```
public function run()
{
    return $this->render('hello');
}
```

By default, views for a widget should be stored in files in the `WidgetPath/views` directory, where `WidgetPath` stands for the directory containing the widget class file. Therefore, the above example will render the view file `@app/components/views/hello.php`, assuming the widget class is located under `@app/components`. You may override the [[yii\base\Widget::getViewPath()]] method to customize the directory containing the widget view files.

Best Practices

Widgets are an object-oriented way of reusing view code.

When creating widgets, you should still follow the MVC pattern. In general, you should keep logic in widget classes and keep presentation in views.

Widgets should be designed to be self-contained. That is, when using a widget, you should be able to just drop it in a view without doing anything else. This could be tricky if a widget requires external resources, such as CSS, JavaScript, images, etc. Fortunately, Yii provides the support for asset bundles, which can be utilized to solve the problem.

When a widget contains view code only, it is very similar to a view. In fact, in this case, their only difference is that a widget is a redistributable class, while a view is just a plain PHP script that you would prefer to keep within your application.

Assets

An asset in Yii is a file that may be referenced in a Web page. It can be a CSS file, a JavaScript file, an image or video file, etc. Assets are located in Web-accessible directories and are directly served by Web servers.

It is often preferable to manage assets programmatically. For example, when you use the [[yii\jui\DatePicker]] widget in a page, it will automatically include the required CSS and JavaScript files, instead of asking you to manually find these files and include them. And when you upgrade the widget to a new version, it will automatically use the new version of the asset files. In this tutorial, we will describe the powerful asset management capability provided in Yii.

Asset Bundles

Yii manages assets in the unit of *asset bundle*. An asset bundle is simply a collection of assets located in a directory. When you register an asset bundle in a view, it will include the CSS and JavaScript files in the bundle in the rendered Web page.

Defining Asset Bundles

Asset bundles are specified as PHP classes extending from [[yii\web\AssetBundle]]. The name of a bundle is simply its corresponding fully qualified PHP class name (without the leading backslash). An asset bundle class should be autoloadable. It usually specifies where the assets are located, what CSS and JavaScript files the bundle contains, and how the bundle depends on other bundles.

The following code defines the main asset bundle used by the basic project template:

```php
<?php

namespace app\assets;

use yii\web\AssetBundle;

class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.css',
    ];
    public $js = [
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

The above `AppAsset` class specifies that the asset files are located under the `@webroot` directory which corresponds to the URL `@web` ; the bundle contains a single CSS file `css/site.css` and no JavaScript file; the bundle depends on two other bundles: [[yii\web\YiiAsset]] and [[yii\bootstrap\BootstrapAsset]]. More detailed explanation about the properties of [[yii\web\AssetBundle]] can be found in the following:

- [[yii\web\AssetBundle::sourcePath|sourcePath]]: specifies the root directory that contains the asset files in this bundle. This property should be set if the root directory is not Web accessible. Otherwise, you should set the [[yii\web\AssetBundle::basePath|basePath]] property and [[yii\web\AssetBundle::baseUrl|baseUrl]], instead. Path aliases can be used here.
- [[yii\web\AssetBundle::basePath|basePath]]: specifies a Web-accessible directory that contains the asset files in this bundle. When you specify the [[yii\web\AssetBundle::sourcePath|sourcePath]] property, the asset manager will publish the assets in this bundle to a Web-accessible directory and overwrite this property accordingly. You should set this property if your asset files are already in a Web-accessible directory and do not need asset publishing. Path aliases can be used here.
- [[yii\web\AssetBundle::baseUrl|baseUrl]]: specifies the URL corresponding to the directory [[yii\web\AssetBundle::basePath|basePath]]. Like [[yii\web\AssetBundle::basePath|basePath]], if you specify the [[yii\web\AssetBundle::sourcePath|sourcePath]] property, the asset manager will publish the assets and overwrite this property accordingly. Path aliases can be used here.
- [[yii\web\AssetBundle::js|js]]: an array listing the JavaScript files contained in this bundle. Note that only forward slash "/" should be used as directory separators. Each

JavaScript file can be specified in one of the following two formats:

  - a relative path representing a local JavaScript file (e.g. `js/main.js`). The actual path of the file can be determined by prepending [[yii\web\AssetManager::basePath]] to the relative path, and the actual URL of the file can be determined by prepending [[yii\web\AssetManager::baseUrl]] to the relative path.
  - an absolute URL representing an external JavaScript file. For example, `http://ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js` or `//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js`.

- [[yii\web\AssetBundle::css|css]]: an array listing the CSS files contained in this bundle. The format of this array is the same as that of [[yii\web\AssetBundle::js|js]].
- [[yii\web\AssetBundle::depends|depends]]: an array listing the names of the asset bundles that this bundle depends on (to be explained shortly).
- [[yii\web\AssetBundle::jsOptions|jsOptions]]: specifies the options that will be passed to the [[yii\web\View::registerJsFile()]] method when it is called to register *every* JavaScript file in this bundle.
- [[yii\web\AssetBundle::cssOptions|cssOptions]]: specifies the options that will be passed to the [[yii\web\View::registerCssFile()]] method when it is called to register *every* CSS file in this bundle.
- [[yii\web\AssetBundle::publishOptions|publishOptions]]: specifies the options that will be passed to the [[yii\web\AssetManager::publish()]] method when it is called to publish source asset files to a Web directory. This is only used if you specify the [[yii\web\AssetBundle::sourcePath|sourcePath]] property.

Asset Locations

Assets, based on their location, can be classified as:

- source assets: the asset files are located together with PHP source code which cannot be directly accessed via Web. In order to use source assets in a page, they should be copied to a Web directory and turned into the so-called published assets. This process is called *asset publishing* which will be described in detail shortly.
- published assets: the asset files are located in a Web directory and can thus be directly accessed via Web.
- external assets: the asset files are located on a Web server that is different from the one hosting your Web application.

When defining an asset bundle class, if you specify the [[yii\web\AssetBundle::sourcePath|sourcePath]] property, it means any assets listed using relative paths will be considered as source assets. If you do not specify this property, it

means those assets are published assets (you should therefore specify [[yii\web\AssetBundle::basePath|basePath]] and [[yii\web\AssetBundle::baseUrl|baseUrl]] to let Yii know where they are located).

It is recommended that you place assets belonging to an application in a Web directory to avoid the unnecessary asset publishing process. This is why `AppAsset` in the prior example specifies [[yii\web\AssetBundle::basePath|basePath]] instead of [[yii\web\AssetBundle::sourcePath|sourcePath]].

For extensions, because their assets are located together with their source code in directories that are not Web accessible, you have to specify the [[yii\web\AssetBundle::sourcePath|sourcePath]] property when defining asset bundle classes for them.

> Note: Do not use `@webroot/assets` as the [[yii\web\AssetBundle::sourcePath|source path]]. This directory is used by default by the [[yii\web\AssetManager|asset manager]] to save the asset files published from their source location. Any content in this directory is considered temporarily and may be subject to removal.

Asset Dependencies

When you include multiple CSS or JavaScript files in a Web page, they have to follow a certain order to avoid overriding issues. For example, if you are using a jQuery UI widget in a Web page, you have to make sure the jQuery JavaScript file is included before the jQuery UI JavaScript file. We call such ordering the dependencies among assets.

Asset dependencies are mainly specified through the [[yii\web\AssetBundle::depends]] property. In the `AppAsset` example, the asset bundle depends on two other asset bundles: [[yii\web\YiiAsset]] and [[yii\bootstrap\BootstrapAsset]], which means the CSS and JavaScript files in `AppAsset` will be included *after* those files in the two dependent bundles.

Asset dependencies are transitive. This means if bundle A depends on B which depends on C, A will depend on C, too.

Asset Options

You can specify the [[yii\web\AssetBundle::cssOptions|cssOptions]] and [[yii\web\AssetBundle::jsOptions|jsOptions]] properties to customize the way that CSS and JavaScript files are included in a page. The values of these properties will be passed to the [[yii\web\View::registerCssFile()]] and [[yii\web\View::registerJsFile()]] methods, respectively, when they are called by the view to include CSS and JavaScript files.

> Note: The options you set in a bundle class apply to *every* CSS/JavaScript file in the bundle. If you want to use different options for different files, you should create separate asset bundles, and use one set of options in each bundle.

For example, to conditionally include a CSS file for browsers that are IE9 or below, you can use the following option:

```
public $cssOptions = ['condition' => 'lte IE9'];
```

This will cause a CSS file in the bundle to be included using the following HTML tags:

```
<!--[if lte IE9]>
<link rel="stylesheet" href="path/to/foo.css">
<![endif]-->
```

To wrap the generated CSS link tags within `<noscript>`, you can configure `cssOptions` as follows,

```
public $cssOptions = ['noscript' => true];
```

To include a JavaScript file in the head section of a page (by default, JavaScript files are included at the end of the body section), use the following option:

```
public $jsOptions = ['position' => \yii\web\View::POS_HEAD];
```

By default, when an asset bundle is being published, all contents in the directory specified by [[yii\web\AssetBundle::sourcePath]] will be published. You can customize this behavior by configuring the [[yii\web\AssetBundle::publishOptions|publishOptions]] property. For example, to publish only one or a few subdirectories of [[yii\web\AssetBundle::sourcePath]], you can do the following in the asset bundle class:

```php
<?php
namespace app\assets;

use yii\web\AssetBundle;

class FontAwesomeAsset extends AssetBundle
{
    public $sourcePath = '@bower/font-awesome';
    public $css = [
        'css/font-awesome.min.css',
    ];
    public $publishOptions = [
        'only' => [
            'fonts/',
            'css/',
        ]
    ];
}
```

The above example defines an asset bundle for the "fontawesome" package. By specifying the `only` publishing option, only the `fonts` and `css` subdirectories will be published.

Bower and NPM Assets

Most JavaScript/CSS packages are managed by Bower and/or NPM. If your application or extension is using such a package, it is recommended that you follow these steps to manage the assets in the library:

1. Modify the `composer.json` file of your application or extension and list the package in the `require` entry. You should use `bower-asset/PackageName` (for Bower packages) or `npm-asset/PackageName` (for NPM packages) to refer to the library.
2. Create an asset bundle class and list the JavaScript/CSS files that you plan to use in your application or extension. You should specify the [[yii\web\AssetBundle::sourcePath|sourcePath]] property as `@bower/PackageName` or `@npm/PackageName`. This is because Composer will install the Bower or NPM package in the directory corresponding to this alias.

> Note: Some packages may put all their distributed files in a subdirectory. If this is the case, you should specify the subdirectory as the value of [[yii\web\AssetBundle::sourcePath|sourcePath]]. For example, [[yii\web\JqueryAsset]] uses `@bower/jquery/dist` instead of `@bower/jquery`.

Using Asset Bundles

To use an asset bundle, register it with a view by calling the [[yii\web\AssetBundle::register()]] method. For example, in a view template you can register an asset bundle like the following:

```
use app\assets\AppAsset;
AppAsset::register($this);  // $this represents the view object
```

> Info: The [[yii\web\AssetBundle::register()]] method returns an asset bundle object containing the information about the published assets, such as [[yii\web\AssetBundle::basePath|basePath]] or [[yii\web\AssetBundle::baseUrl|baseUrl]].

If you are registering an asset bundle in other places, you should provide the needed view object. For example, to register an asset bundle in a widget class, you can get the view object by `$this->view`.

When an asset bundle is registered with a view, behind the scenes Yii will register all its dependent asset bundles. And if an asset bundle is located in a directory inaccessible through the Web, it will be published to a Web directory. Later, when the view renders a page, it will generate `<link>` and `<script>` tags for the CSS and JavaScript files listed in the registered bundles. The order of these tags is determined by the dependencies among the registered bundles and the order of the assets listed in the [[yii\web\AssetBundle::css]] and [[yii\web\AssetBundle::js]] properties.

Dynamic Asset Bundles

Being a regular PHP class asset bundle can bear some extra logic related to it and may adjust its internal parameters dynamically. For example: you may use som sophisticated JavaScript library, which provides some internationalization packed in separated source files: each per each supported language. Thus you will need to add particular '.js' file to your page in order to make library translation work. This can be achieved overriding [[yii\web\AssetBundle::init()]] method:

```
namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class SophisticatedAssetBundle extends AssetBundle
{
    public $sourcePath = '/path/to/sophisticated/src';
    public $js = [
        'sophisticated.js' // file, which is always used
    ];

    public function init()
    {
        parent::init();
        $this->js[] = 'i18n/' . Yii::$app->language . '.js'; // dynamic file added
    }
}
```

Particular asset bundle can also be adjusted via its instance returned by [[yii\web\AssetBundle::register()]]. For example:

```
use app\assets\SophisticatedAssetBundle;
use Yii;

$bundle = SophisticatedAssetBundle::register(Yii::$app->view);
$bundle->js[] = 'i18n/' . Yii::$app->language . '.js'; // dynamic file added
```

> Note: although dynamic adjustment of the asset bundles is supported, it is a **bad** practice, which may lead to unexpected side effects, and should be avoided if possible.

Customizing Asset Bundles

Yii manages asset bundles through an application component named `assetManager` which is implemented by [[yii\web\AssetManager]]. By configuring the [[yii\web\AssetManager::bundles]] property, it is possible to customize the behavior of an asset bundle. For example, the default [[yii\web\JqueryAsset]] asset bundle uses the `jquery.js` file from the installed jquery Bower package. To improve the availability and performance, you may want to use a version hosted by Google. This can be achieved by configuring `assetManager` in the application configuration like the following:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => [
                    'sourcePath' => null,   // do not publish the bundle
                    'js' => [
                        '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.min.js',
                    ]
                ],
            ],
        ],
    ],
];
```

You can configure multiple asset bundles similarly through [[yii\web\AssetManager::bundles]]. The array keys should be the class names (without the leading backslash) of the asset bundles, and the array values should be the corresponding configuration arrays.

> Tip: You can conditionally choose which assets to use in an asset bundle. The following example shows how to use `jquery.js` in the development environment and `jquery.min.js` otherwise:
>
> ```
> 'yii\web\JqueryAsset' => [
>     'js' => [
>         YII_ENV_DEV ? 'jquery.js' : 'jquery.min.js'
>     ]
> ],
> ```

You can disable one or multiple asset bundles by associating `false` with the names of the asset bundles that you want to disable. When you register a disabled asset bundle with a view, none of its dependent bundles will be registered, and the view also will not include any of the assets in the bundle in the page it renders. For example, to disable [[yii\web\JqueryAsset]], you can use the following configuration:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => false,
            ],
        ],
    ],
];
```

You can also disable *all* asset bundles by setting [[yii\web\AssetManager::bundles]] as `false`.

Keep in mind that customization made via [[yii\web\AssetManager::bundles]] is applied at the creation of the asset bundle, e.g. at object constructor stage. Thus any adjustments made to the bundle object after that will override the mapping setup at [[yii\web\AssetManager::bundles]] level. In particular: adjustments made inside [[yii\web\AssetBundle::init()]] method or over the registered bundle object will take precedence over `AssetManager` configuration. Here are the examples, where mapping set via [[yii\web\AssetManager::bundles]] makes no effect:

```
// Program source code:

namespace app\assets;

use yii\web\AssetBundle;
use Yii;

class LanguageAssetBundle extends AssetBundle
{
    // ...

    public function init()
    {
        parent::init();
        $this->baseUrl = '@web/i18n/' . Yii::$app->language; // can NOT be handled by
`AssetManager`!
    }
}
// ...

$bundle = \app\assets\LargeFileAssetBundle::register(Yii::$app->view);
$bundle->baseUrl = YII_DEBUG ? '@web/large-files': '@web/large-files/minified'; // can
 NOT be handled by `AssetManager`!


// Application config :

return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'app\assets\LanguageAssetBundle' => [
                    'baseUrl' => 'http://some.cdn.com/files/i18n/en' // makes NO effec
t!
                ],
                'app\assets\LargeFileAssetBundle' => [
                    'baseUrl' => 'http://some.cdn.com/files/large-files' // makes NO e
ffect!
                ],
            ],
        ],
    ],
];
```

Asset Mapping

Sometimes you may want to "fix" incorrect/incompatible asset file paths used in multiple asset bundles. For example, bundle A uses `jquery.min.js` version 1.11.1, and bundle B uses `jquery.js` version 2.1.1. While you can fix the problem by customizing each bundle,

an easier way is to use the *asset map* feature to map incorrect assets to the desired ones. To do so, configure the [[yii\web\AssetManager::assetMap]] property like the following:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'assetMap' => [
                'jquery.js' => '//ajax.googleapis.com/ajax/libs/jquery/2.1.1/jquery.mi
n.js',
            ],
        ],
    ],
];
```

The keys of [[yii\web\AssetManager::assetMap|assetMap]] are the asset names that you want to fix, and the values are the desired asset paths. When you register an asset bundle with a view, each relative asset file in its [[yii\web\AssetBundle::css|css]] and [[yii\web\AssetBundle::js|js]] arrays will be examined against this map. If any of the keys are found to be the last part of an asset file (which is prefixed with [[yii\web\AssetBundle::sourcePath]] if available), the corresponding value will replace the asset and be registered with the view. For example, the asset file `my/path/to/jquery.js` matches the key `jquery.js`.

> Note: Only assets specified using relative paths are subject to asset mapping. The target asset paths should be either absolute URLs or paths relative to [[yii\web\AssetManager::basePath]].

Asset Publishing

As aforementioned, if an asset bundle is located in a directory that is not Web accessible, its assets will be copied to a Web directory when the bundle is being registered with a view. This process is called *asset publishing*, and is done automatically by the [[yii\web\AssetManager|asset manager]].

By default, assets are published to the directory `@webroot/assets` which corresponds to the URL `@web/assets`. You may customize this location by configuring the [[yii\web\AssetManager::basePath|basePath]] and [[yii\web\AssetManager::baseUrl|baseUrl]] properties.

Instead of publishing assets by file copying, you may consider using symbolic links, if your OS and Web server allow. This feature can be enabled by setting [[yii\web\AssetManager::linkAssets|linkAssets]] to be `true`.

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'linkAssets' => true,
        ],
    ],
];
```

With the above configuration, the asset manager will create a symbolic link to the source path of an asset bundle when it is being published. This is faster than file copying and can also ensure that the published assets are always up-to-date.

Cache Busting

For Web application running in production mode, it is a common practice to enable HTTP caching for assets and other static resources. A drawback of this practice is that whenever you modify an asset and deploy it to production, a user client may still use the old version due to the HTTP caching. To overcome this drawback, you may use the cache busting feature, which was introduced in version 2.0.3, by configuring [[yii\web\AssetManager]] like the following:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'appendTimestamp' => true,
        ],
    ],
];
```

By doing so, the URL of every published asset will be appended with its last modification timestamp. For example, the URL to `yii.js` may look like `/assets/5515a87c/yii.js?v=1423448645"`, where the parameter `v` represents the last modification timestamp of the `yii.js` file. Now if you modify an asset, its URL will be changed, too, which causes the client to fetch the latest version of the asset.

Commonly Used Asset Bundles

The core Yii code has defined many asset bundles. Among them, the following bundles are commonly used and may be referenced in your application or extension code.

- [[yii\web\YiiAsset]]: It mainly includes the `yii.js` file which implements a mechanism of

organizing JavaScript code in modules. It also provides special support for `data-method` and `data-confirm` attributes and other useful features. More information about `yii.js` can be found in the Client Scripts Section.
- [[yii\web\JqueryAsset]]: It includes the `jquery.js` file from the jQuery Bower package.
- [[yii\bootstrap\BootstrapAsset]]: It includes the CSS file from the Twitter Bootstrap framework.
- [[yii\bootstrap\BootstrapPluginAsset]]: It includes the JavaScript file from the Twitter Bootstrap framework for supporting Bootstrap JavaScript plugins.
- [[yii\jui\JuiAsset]]: It includes the CSS and JavaScript files from the jQuery UI library.

If your code depends on jQuery, jQuery UI or Bootstrap, you should use these predefined asset bundles rather than creating your own versions. If the default setting of these bundles do not satisfy your needs, you may customize them as described in the Customizing Asset Bundle subsection.

Asset Conversion

Instead of directly writing CSS and/or JavaScript code, developers often write them in some extended syntax and use special tools to convert it into CSS/JavaScript. For example, for CSS code you may use LESS or SCSS; and for JavaScript you may use TypeScript.

You can list the asset files in extended syntax in the [[yii\web\AssetBundle::css|css]] and [[yii\web\AssetBundle::js|js]] properties of an asset bundle. For example,

```
class AppAsset extends AssetBundle
{
    public $basePath = '@webroot';
    public $baseUrl = '@web';
    public $css = [
        'css/site.less',
    ];
    public $js = [
        'js/site.ts',
    ];
    public $depends = [
        'yii\web\YiiAsset',
        'yii\bootstrap\BootstrapAsset',
    ];
}
```

When you register such an asset bundle with a view, the [[yii\web\AssetManager|asset manager]] will automatically run the pre-processor tools to convert assets in recognized extended syntax into CSS/JavaScript. When the view finally renders a page, it will include the CSS/JavaScript files in the page, instead of the original assets in extended syntax.

Yii uses the file name extensions to identify which extended syntax an asset is in. By default it recognizes the following syntax and file name extensions:

- LESS: `.less`
- SCSS: `.scss`
- Stylus: `.styl`
- CoffeeScript: `.coffee`
- TypeScript: `.ts`

Yii relies on the installed pre-processor tools to convert assets. For example, to use LESS you should install the `lessc` pre-processor command.

You can customize the pre-processor commands and the supported extended syntax by configuring [[yii\web\AssetManager::converter]] like the following:

```
return [
    'components' => [
        'assetManager' => [
            'converter' => [
                'class' => 'yii\web\AssetConverter',
                'commands' => [
                    'less' => ['css', 'lessc {from} {to} --no-color'],
                    'ts' => ['js', 'tsc --out {to} {from}'],
                ],
            ],
        ],
    ],
];
```

In the above, we specify the supported extended syntax via the [[yii\web\AssetConverter::commands]] property. The array keys are the file extension names (without leading dot), and the array values are the resulting asset file extension names and the commands for performing the asset conversion. The tokens `{from}` and `{to}` in the commands will be replaced with the source asset file paths and the target asset file paths.

> Info: There are other ways of working with assets in extended syntax, besides the one described above. For example, you can use build tools such as grunt to monitor and automatically convert assets in extended syntax. In this case, you should list the resulting CSS/JavaScript files in asset bundles rather than the original files.

Combining and Compressing Assets

A Web page can include many CSS and/or JavaScript files. To reduce the number of HTTP requests and the overall download size of these files, a common practice is to combine and compress multiple CSS/JavaScript files into one or very few files, and then include these compressed files instead of the original ones in the Web pages.

> Info: Combining and compressing assets are usually needed when an application is in production mode. In development mode, using the original CSS/JavaScript files is often more convenient for debugging purposes.

In the following, we introduce an approach to combine and compress asset files without the need to modify your existing application code.

1. Find all the asset bundles in your application that you plan to combine and compress.
2. Divide these bundles into one or a few groups. Note that each bundle can only belong to a single group.
3. Combine/compress the CSS files in each group into a single file. Do this similarly for the JavaScript files.
4. Define a new asset bundle for each group:
    - Set the [[yii\web\AssetBundle::css|css]] and [[yii\web\AssetBundle::js|js]] properties to be the combined CSS and JavaScript files, respectively.
    - Customize the asset bundles in each group by setting their [[yii\web\AssetBundle::css|css]] and [[yii\web\AssetBundle::js|js]] properties to be empty, and setting their [[yii\web\AssetBundle::depends|depends]] property to be the new asset bundle created for the group.

Using this approach, when you register an asset bundle in a view, it causes the automatic registration of the new asset bundle for the group that the original bundle belongs to. And as a result, the combined/compressed asset files are included in the page, instead of the original ones.

An Example

Let's use an example to further explain the above approach.

Assume your application has two pages, X and Y. Page X uses asset bundles A, B and C, while Page Y uses asset bundles B, C and D.

You have two ways to divide these asset bundles. One is to use a single group to include all asset bundles, the other is to put A in Group X, D in Group Y, and (B, C) in Group S. Which one is better? It depends. The first way has the advantage that both pages share the same combined CSS and JavaScript files, which makes HTTP caching more effective. On the other hand, because the single group contains all bundles, the size of the combined CSS

and JavaScript files will be bigger and thus increase the initial file transmission time. For simplicity in this example, we will use the first way, i.e., use a single group to contain all bundles.

> Info: Dividing asset bundles into groups is not trivial task. It usually requires analysis about the real world traffic data of various assets on different pages. At the beginning, you may start with a single group for simplicity.

Use existing tools (e.g. Closure Compiler, YUI Compressor) to combine and compress CSS and JavaScript files in all the bundles. Note that the files should be combined in the order that satisfies the dependencies among the bundles. For example, if Bundle A depends on B which depends on both C and D, then you should list the asset files starting from C and D, followed by B and finally A.

After combining and compressing, we get one CSS file and one JavaScript file. Assume they are named as `all-xyz.css` and `all-xyz.js`, where `xyz` stands for a timestamp or a hash that is used to make the file name unique to avoid HTTP caching problems.

We are at the last step now. Configure the [[yii\web\AssetManager|asset manager]] as follows in the application configuration:

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => [
                'all' => [
                    'class' => 'yii\web\AssetBundle',
                    'basePath' => '@webroot/assets',
                    'baseUrl' => '@web/assets',
                    'css' => ['all-xyz.css'],
                    'js' => ['all-xyz.js'],
                ],
                'A' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'B' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'C' => ['css' => [], 'js' => [], 'depends' => ['all']],
                'D' => ['css' => [], 'js' => [], 'depends' => ['all']],
            ],
        ],
    ],
];
```

As explained in the Customizing Asset Bundles subsection, the above configuration changes the default behavior of each bundle. In particular, Bundle A, B, C and D no longer have any asset files. They now all depend on the `all` bundle which contains the combined `all-`

`xyz.css` and `all-xyz.js` files. Consequently, for Page X, instead of including the original source files from Bundle A, B and C, only these two combined files will be included; the same thing happens to Page Y.

There is one final trick to make the above approach work more smoothly. Instead of directly modifying the application configuration file, you may put the bundle customization array in a separate file and conditionally include this file in the application configuration. For example,

```
return [
    'components' => [
        'assetManager' => [
            'bundles' => require(__DIR__ . '/' . (YII_ENV_PROD ? 'assets-prod.php' : '
assets-dev.php')),
        ],
    ],
];
```

That is, the asset bundle configuration array is saved in `assets-prod.php` for production mode, and `assets-dev.php` for non-production mode.

> Note: this asset combining mechanism is based on the ability of [[yii\web\AssetManager::bundles]] to override the properties of the registered asset bundles. However, as it already has been said above, this ability does not cover asset bundle adjustments, which are performed at [[yii\web\AssetBundle::init()]] method or after bundle is registered. You should avoid usage of such dynamic bundles during the asset combining.

Using the `asset` Command

Yii provides a console command named `asset` to automate the approach that we just described.

To use this command, you should first create a configuration file to describe what asset bundles should be combined and how they should be grouped. You can use the `asset/template` sub-command to generate a template first and then modify it to fit for your needs.

```
yii asset/template assets.php
```

The command generates a file named `assets.php` in the current directory. The content of this file looks like the following:

```
<?php
/**
 * Configuration file for the "yii asset" console command.
 * Note that in the console environment, some path aliases like '@webroot' and '@web'
may not exist.
 * Please define these missing path aliases.
 */
return [
    // Adjust command/callback for JavaScript files compressing:
    'jsCompressor' => 'java -jar compiler.jar --js {from} --js_output_file {to}',
    // Adjust command/callback for CSS files compressing:
    'cssCompressor' => 'java -jar yuicompressor.jar --type css {from} -o {to}',
    // Whether to delete asset source after compression:
    'deleteSource' => false,
    // The list of asset bundles to compress:
    'bundles' => [
        // 'yii\web\YiiAsset',
        // 'yii\web\JqueryAsset',
    ],
    // Asset bundle for compression output:
    'targets' => [
        'all' => [
            'class' => 'yii\web\AssetBundle',
            'basePath' => '@webroot/assets',
            'baseUrl' => '@web/assets',
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
        ],
    ],
    // Asset manager configuration:
    'assetManager' => [
    ],
];
```

You should modify this file and specify which bundles you plan to combine in the `bundles` option. In the `targets` option you should specify how the bundles should be divided into groups. You can specify one or multiple groups, as aforementioned.

> Note: Because the alias `@webroot` and `@web` are not available in the console application, you should explicitly define them in the configuration.

JavaScript files are combined, compressed and written to `js/all-{hash}.js` where {hash} is replaced with the hash of the resulting file.

The `jsCompressor` and `cssCompressor` options specify the console commands or PHP callbacks for performing JavaScript and CSS combining/compressing. By default, Yii uses Closure Compiler for combining JavaScript files and YUI Compressor for combining CSS files. You should install those tools manually or adjust these options to use your favorite tools.

With the configuration file, you can run the `asset` command to combine and compress the asset files and then generate a new asset bundle configuration file `assets-prod.php`:

```
yii asset assets.php config/assets-prod.php
```

The generated configuration file can be included in the application configuration, like described in the last subsection.

> Note: in case you customize asset bundles for your application via [[yii\web\AssetManager::bundles]] or [[yii\web\AssetManager::assetMap]] and want this customization to be applied for the compression source files, you should include these options to the `assetManager` section inside asset command configuration file.
>
> Note: while specifying the compression source, you should avoid the use of asset bundles whose parameters may be adjusted dynamically (e.g. at `init()` method or after registration), since they may work incorrectly after compression.
>
> Info: Using the `asset` command is not the only option to automate the asset combining and compressing process. You can use the excellent task runner tool grunt to achieve the same goal.

Grouping Asset Bundles

In the last subsection, we have explained how to combine all asset bundles into a single one in order to minimize the HTTP requests for asset files referenced in an application. This is not always desirable in practice. For example, imagine your application has a "front end" as well as a "back end", each of which uses a different set of JavaScript and CSS files. In this case, combining all asset bundles from both ends into a single one does not make sense, because the asset bundles for the "front end" are not used by the "back end" and it would be a waste of network bandwidth to send the "back end" assets when a "front end" page is requested.

To solve the above problem, you can divide asset bundles into groups and combine asset bundles for each group. The following configuration shows how you can group asset bundles:

```
return [
    ...
    // Specify output bundles with groups:
    'targets' => [
        'allShared' => [
            'js' => 'js/all-shared-{hash}.js',
            'css' => 'css/all-shared-{hash}.css',
            'depends' => [
                // Include all assets shared between 'backend' and 'frontend'
                'yii\web\YiiAsset',
                'app\assets\SharedAsset',
            ],
        ],
        'allBackEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [
                // Include only 'backend' assets:
                'app\assets\AdminAsset'
            ],
        ],
        'allFrontEnd' => [
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
            'depends' => [], // Include all remaining assets
        ],
    ],
    ...
];
```

As you can see, the asset bundles are divided into three groups: `allShared`, `allBackEnd` and `allFrontEnd`. They each depends on an appropriate set of asset bundles. For example, `allBackEnd` depends on `app\assets\AdminAsset`. When running `asset` command with this configuration, it will combine asset bundles according to the above specification.

> Info: You may leave the `depends` configuration empty for one of the target bundle. By doing so, that particular asset bundle will depend on all of the remaining asset bundles that other target bundles do not depend on.

Extensions

Extensions are redistributable software packages specifically designed to be used in Yii applications and provide ready-to-use features. For example, the yiisoft/yii2-debug extension adds a handy debug toolbar at the bottom of every page in your application to help you more easily grasp how the pages are generated. You can use extensions to accelerate your development process. You can also package your code as extensions to share with other people your great work.

> Info: We use the term "extension" to refer to Yii-specific software packages. For general purpose software packages that can be used without Yii, we will refer to them using the term "package" or "library".

Using Extensions

To use an extension, you need to install it first. Most extensions are distributed as Composer packages which can be installed by taking the following two simple steps:

1. modify the `composer.json` file of your application and specify which extensions (Composer packages) you want to install.
2. run `composer install` to install the specified extensions.

Note that you may need to install Composer if you do not have it.

By default, Composer installs packages registered on Packagist - the biggest repository for open source Composer packages. You can look for extensions on Packagist. You may also create your own repository and configure Composer to use it. This is useful if you are developing private extensions that you want to share within your projects only.

Extensions installed by Composer are stored in the `BasePath/vendor` directory, where `BasePath` refers to the application's base path. Because Composer is a dependency manager, when it installs a package, it will also install all its dependent packages.

For example, to install the `yiisoft/yii2-imagine` extension, modify your `composer.json` like the following:

```
{
    // ...

    "require": {
        // ... other dependencies

        "yiisoft/yii2-imagine": "*"
    }
}
```

After the installation, you should see the directory `yiisoft/yii2-imagine` under `BasePath/vendor`. You should also see another directory `imagine/imagine` which contains the installed dependent package.

> Info: The `yiisoft/yii2-imagine` is a core extension developed and maintained by the Yii developer team. All core extensions are hosted on Packagist and named like `yiisoft/yii2-xyz`, where `xyz` varies for different extensions.

Now you can use the installed extensions like they are part of your application. The following example shows how you can use the `yii\imagine\Image` class provided by the `yiisoft/yii2-imagine` extension:

```
use Yii;
use yii\imagine\Image;

// generate a thumbnail image
Image::thumbnail('@webroot/img/test-image.jpg', 120, 120)
    ->save(Yii::getAlias('@runtime/thumb-test-image.jpg'), ['quality' => 50]);
```

> Info: Extension classes are autoloaded by the Yii class autoloader.

Installing Extensions Manually

In some rare occasions, you may want to install some or all extensions manually, rather than relying on Composer. To do so, you should:

1. download the extension archive files and unpack them in the `vendor` directory.
2. install the class autoloaders provided by the extensions, if any.
3. download and install all dependent extensions as instructed.

If an extension does not have a class autoloader but follows the PSR-4 standard, you may use the class autoloader provided by Yii to autoload the extension classes. All you need to do is just to declare a root alias for the extension root directory. For example, assuming you

have installed an extension in the directory `vendor/mycompany/myext`, and the extension classes are under the `myext` namespace, then you can include the following code in your application configuration:

```
[
    'aliases' => [
        '@myext' => '@vendor/mycompany/myext',
    ],
]
```

Creating Extensions

You may consider creating an extension when you feel the need to share with other people your great code. An extension can contain any code you like, such as a helper class, a widget, a module, etc.

It is recommended that you create an extension in terms of a Composer package so that it can be more easily installed and used by other users, as described in the last subsection.

Below are the basic steps you may follow to create an extension as a Composer package.

1. Create a project for your extension and host it on a VCS repository, such as github.com. The development and maintenance work for the extension should be done on this repository.
2. Under the root directory of the project, create a file named `composer.json` as required by Composer. Please refer to the next subsection for more details.
3. Register your extension with a Composer repository, such as Packagist, so that other users can find and install your extension using Composer.

`composer.json`

Each Composer package must have a `composer.json` file in its root directory. The file contains the metadata about the package. You may find complete specification about this file in the Composer Manual. The following example shows the `composer.json` file for the `yiisoft/yii2-imagine` extension:

```
{
    // package name
    "name": "yiisoft/yii2-imagine",

    // package type
    "type": "yii2-extension",

    "description": "The Imagine integration for the Yii framework",
    "keywords": ["yii2", "imagine", "image", "helper"],
    "license": "BSD-3-Clause",
    "support": {
        "issues": "https://github.com/yiisoft/yii2/issues?labels=ext%3Aimagine",
        "forum": "http://www.yiiframework.com/forum/",
        "wiki": "http://www.yiiframework.com/wiki/",
        "irc": "irc://irc.freenode.net/yii",
        "source": "https://github.com/yiisoft/yii2"
    },
    "authors": [
        {
            "name": "Antonio Ramirez",
            "email": "amigo.cobos@gmail.com"
        }
    ],

    // package dependencies
    "require": {
        "yiisoft/yii2": "~2.0.0",
        "imagine/imagine": "v0.5.0"
    },

    // class autoloading specs
    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

Package Name

Each Composer package should have a package name which uniquely identifies the package among all others. The format of package names is `vendorName/projectName`. For example, in the package name `yiisoft/yii2-imagine`, the vendor name and the project name are `yiisoft` and `yii2-imagine`, respectively.

Do NOT use `yiisoft` as your vendor name as it is reserved for use by the Yii core code.

We recommend you prefix `yii2-` to the project name for packages representing Yii 2 extensions, for example, `myname/yii2-mywidget`. This will allow users to more easily tell whether a package is a Yii 2 extension.

Package Type

It is important that you specify the package type of your extension as `yii2-extension` so that the package can be recognized as a Yii extension when being installed.

When a user runs `composer install` to install an extension, the file `vendor/yiisoft/extensions.php` will be automatically updated to include the information about the new extension. From this file, Yii applications can know which extensions are installed (the information can be accessed via [[yii\base\Application::extensions]]).

Dependencies

Your extension depends on Yii (of course). So you should list it (`yiisoft/yii2`) in the `require` entry in `composer.json`. If your extension also depends on other extensions or third-party libraries, you should list them as well. Make sure you also list appropriate version constraints (e.g. `1.*`, `@stable`) for each dependent package. Use stable dependencies when your extension is released in a stable version.

Most JavaScript/CSS packages are managed using Bower and/or NPM, instead of Composer. Yii uses the Composer asset plugin to enable managing these kinds of packages through Composer. If your extension depends on a Bower package, you can simply list the dependency in `composer.json` like the following:

```
{
    // package dependencies
    "require": {
        "bower-asset/jquery": ">=1.11.*"
    }
}
```

The above code states that the extension depends on the `jquery` Bower package. In general, you can use `bower-asset/PackageName` to refer to a Bower package in `composer.json`, and use `npm-asset/PackageName` to refer to a NPM package. When Composer installs a Bower or NPM package, by default the package content will be installed under the `@vendor/bower/PackageName` and `@vendor/npm/Packages` directories, respectively. These two directories can also be referred to using the shorter aliases `@bower/PackageName` and `@npm/PackageName`.

For more details about asset management, please refer to the Assets section.

Class Autoloading

In order for your classes to be autoloaded by the Yii class autoloader or the Composer class autoloader, you should specify the `autoload` entry in the `composer.json` file, like shown below:

```
{
    // ....

    "autoload": {
        "psr-4": {
            "yii\\imagine\\": ""
        }
    }
}
```

You may list one or multiple root namespaces and their corresponding file paths.

When the extension is installed in an application, Yii will create for each listed root namespace an alias that refers to the directory corresponding to the namespace. For example, the above `autoload` declaration will correspond to an alias named `@yii/imagine`.

Recommended Practices

Because extensions are meant to be used by other people, you often need to make an extra effort during development. Below we introduce some common and recommended practices in creating high quality extensions.

Namespaces

To avoid name collisions and make the classes in your extension autoloadable, you should use namespaces and name the classes in your extension by following the PSR-4 standard or PSR-0 standard.

Your class namespaces should start with `vendorName\extensionName`, where `extensionName` is similar to the project name in the package name except that it should not contain the `yii2-` prefix. For example, for the `yiisoft/yii2-imagine` extension, we use `yii\imagine` as the namespace for its classes.

Do not use `yii`, `yii2` or `yiisoft` as your vendor name. These names are reserved for use by the Yii core code.

Bootstrapping Classes

Sometimes, you may want your extension to execute some code during the bootstrapping process stage of an application. For example, your extension may want to respond to the application's `beginRequest` event to adjust some environment settings. While you can instruct users of the extension to explicitly attach your event handler in the extension to the `beginRequest` event, a better way is to do this automatically.

To achieve this goal, you can create a so-called *bootstrapping class* by implementing [[yii\base\BootstrapInterface]]. For example,

```
namespace myname\mywidget;

use yii\base\BootstrapInterface;
use yii\base\Application;

class MyBootstrapClass implements BootstrapInterface
{
    public function bootstrap($app)
    {
        $app->on(Application::EVENT_BEFORE_REQUEST, function () {
            // do something here
        });
    }
}
```

You then list this class in the `composer.json` file of your extension like follows,

```
{
    // ...

    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

When the extension is installed in an application, Yii will automatically instantiate the bootstrapping class and call its [[yii\base\BootstrapInterface::bootstrap()|bootstrap()]] method during the bootstrapping process for every request.

Working with Databases

Your extension may need to access databases. Do not assume that the applications that use your extension will always use `Yii::$db` as the DB connection. Instead, you should declare a `db` property for the classes that require DB access. The property will allow users of your

extension to customize which DB connection they would like your extension to use. As an example, you may refer to the [[yii\caching\DbCache]] class and see how it declares and uses the `db` property.

If your extension needs to create specific DB tables or make changes to DB schema, you should

- provide migrations to manipulate DB schema, rather than using plain SQL files;
- try to make the migrations applicable to different DBMS;
- avoid using Active Record in the migrations.

Using Assets

If your extension is a widget or a module, chances are that it may require some assets to work. For example, a module may display some pages which contain images, JavaScript, and CSS. Because the files of an extension are all under the same directory which is not Web accessible when installed in an application, you have two choices to make the asset files directly accessible via Web:

- ask users of the extension to manually copy the asset files to a specific Web-accessible folder;
- declare an asset bundle and rely on the asset publishing mechanism to automatically copy the files listed in the asset bundle to a Web-accessible folder.

We recommend you use the second approach so that your extension can be more easily used by other people. Please refer to the Assets section for more details about how to work with assets in general.

Internationalization and Localization

Your extension may be used by applications supporting different languages! Therefore, if your extension displays content to end users, you should try to internationalize and localize it. In particular,

- If the extension displays messages intended for end users, the messages should be wrapped into `Yii::t()` so that they can be translated. Messages meant for developers (such as internal exception messages) do not need to be translated.
- If the extension displays numbers, dates, etc., they should be formatted using [[yii\i18n\Formatter]] with appropriate formatting rules.

For more details, please refer to the Internationalization section.

Testing

You want your extension to run flawlessly without bringing problems to other people. To reach this goal, you should test your extension before releasing it to public.

It is recommended that you create various test cases to cover your extension code rather than relying on manual tests. Each time before you release a new version of your extension, you may simply run these test cases to make sure everything is in good shape. Yii provides testing support, which can help you to more easily write unit tests, acceptance tests and functionality tests. For more details, please refer to the Testing section.

Versioning

You should give each release of your extension a version number (e.g. `1.0.1`). We recommend you follow the semantic versioning practice when determining what version numbers should be used.

Releasing

To let other people know about your extension, you need to release it to the public.

If it is the first time you are releasing an extension, you should register it on a Composer repository, such as Packagist. After that, all you need to do is simply create a release tag (e.g. `v1.0.1`) on the VCS repository of your extension and notify the Composer repository about the new release. People will then be able to find the new release, and install or update the extension through the Composer repository.

In the releases of your extension, in addition to code files, you should also consider including the following to help other people learn about and use your extension:

- A readme file in the package root directory: it describes what your extension does and how to install and use it. We recommend you write it in Markdown format and name the file as `readme.md`.
- A changelog file in the package root directory: it lists what changes are made in each release. The file may be written in Markdown format and named as `changelog.md`.
- An upgrade file in the package root directory: it gives the instructions on how to upgrade from older releases of the extension. The file may be written in Markdown format and named as `upgrade.md`.
- Tutorials, demos, screenshots, etc.: these are needed if your extension provides many features that cannot be fully covered in the readme file.
- API documentation: your code should be well documented to allow other people to more easily read and understand it. You may refer to the Object class file to learn how to document your code.

> Info: Your code comments can be written in Markdown format. The `yiisoft/yii2-apidoc` extension provides a tool for you to generate pretty API documentation based on your code comments.
>
> Info: While not a requirement, we suggest your extension adhere to certain coding styles. You may refer to the core framework code style.

Core Extensions

Yii provides the following core extensions that are developed and maintained by the Yii developer team. They are all registered on Packagist and can be easily installed as described in the Using Extensions subsection.

- yiisoft/yii2-apidoc: provides an extensible and high-performance API documentation generator. It is also used to generate the core framework API documentation.
- yiisoft/yii2-authclient: provides a set of commonly used auth clients, such as Facebook OAuth2 client, GitHub OAuth2 client.
- yiisoft/yii2-bootstrap: provides a set of widgets that encapsulate the Bootstrap components and plugins.
- yiisoft/yii2-codeception: provides testing support based on Codeception.
- yiisoft/yii2-debug: provides debugging support for Yii applications. When this extension is used, a debugger toolbar will appear at the bottom of every page. The extension also provides a set of standalone pages to display more detailed debug information.
- yiisoft/yii2-elasticsearch: provides the support for using Elasticsearch. It includes basic querying/search support and also implements the Active Record pattern that allows you to store active records in Elasticsearch.
- yiisoft/yii2-faker: provides the support for using Faker to generate fake data for you.
- yiisoft/yii2-gii: provides a Web-based code generator that is highly extensible and can be used to quickly generate models, forms, modules, CRUD, etc.
- yiisoft/yii2-httpclient: provides an HTTP client.
- yiisoft/yii2-imagine: provides commonly used image manipulation functions based on Imagine.
- yiisoft/yii2-jui: provides a set of widgets that encapsulate the JQuery UI interactions and widgets.
- yiisoft/yii2-mongodb: provides the support for using MongoDB. It includes features such as basic query, Active Record, migrations, caching, code generation, etc.
- yiisoft/yii2-redis: provides the support for using redis. It includes features such as basic query, Active Record, caching, etc.
- yiisoft/yii2-smarty: provides a template engine based on Smarty.
- yiisoft/yii2-sphinx: provides the support for using Sphinx. It includes features such as

basic query, Active Record, code generation, etc.

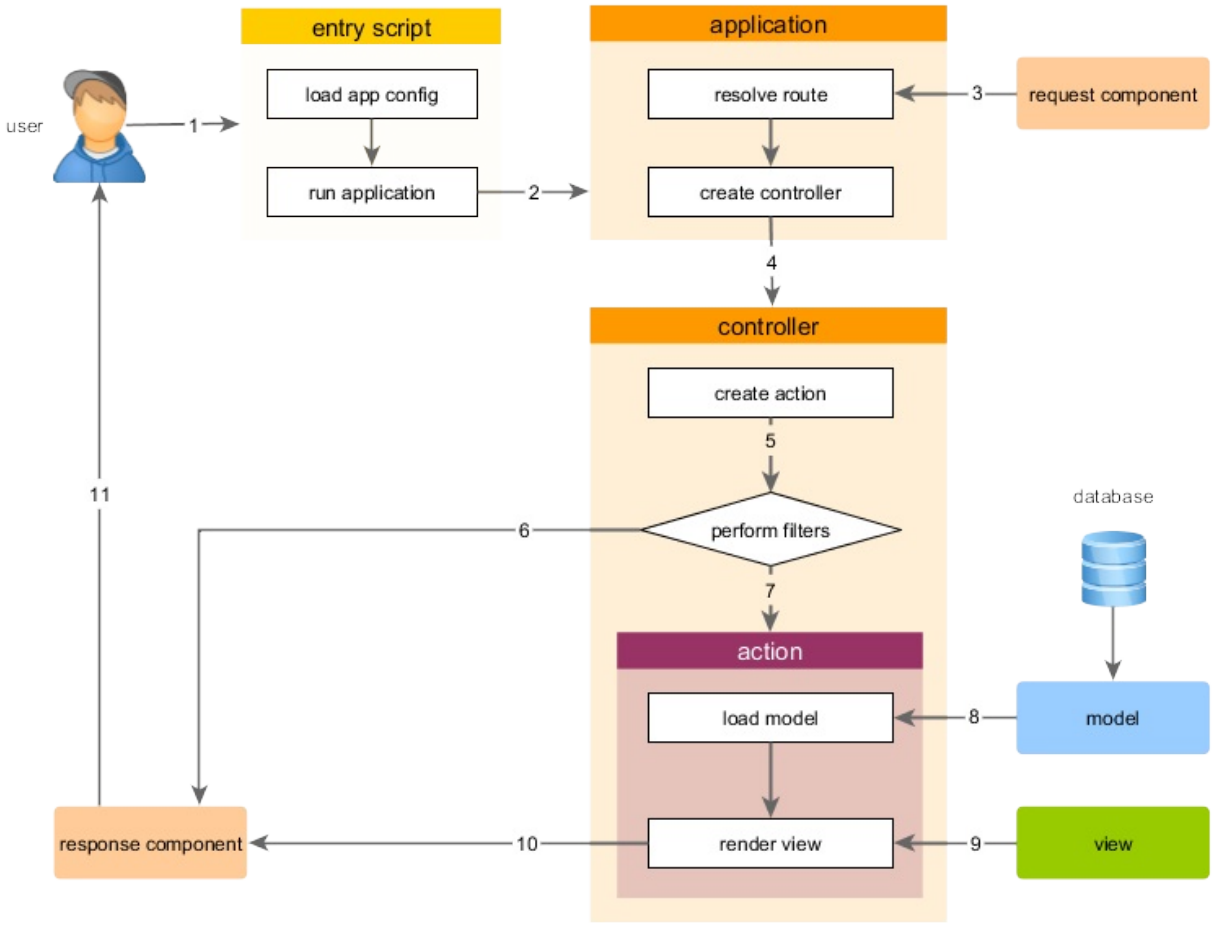
- yiisoft/yii2-swiftmailer: provides email sending features based on swiftmailer.
- yiisoft/yii2-twig: provides a template engine based on Twig.

Overview

Each time when a Yii application handles a request, it undergoes a similar workflow.

1. A user makes a request to the entry script `web/index.php`.
2. The entry script loads the application configuration and creates an application instance to handle the request.
3. The application resolves the requested route with the help of the request application component.
4. The application creates a controller instance to handle the request.
5. The controller creates an action instance and performs the filters for the action.
6. If any filter fails, the action is cancelled.
7. If all filters pass, the action is executed.
8. The action loads a data model, possibly from a database.
9. The action renders a view, providing it with the data model.
10. The rendered result is returned to the response application component.
11. The response component sends the rendered result to the user's browser.

The following diagram shows how an application handles a request.



In this section, we will describe in detail how some of these steps work.

Bootstrapping

Bootstrapping refers to the process of preparing the environment before an application starts to resolve and process an incoming request. Bootstrapping is done in two places: the entry script and the application.

In the entry script, class autoloaders for different libraries are registered. This includes the Composer autoloader through its `autoload.php` file and the Yii autoloader through its `Yii` class file. The entry script then loads the application configuration and creates an application instance.

In the constructor of the application, the following bootstrapping work is done:

1. [[yii\base\Application::preInit()|preInit()]] is called, which configures some high priority application properties, such as [[yii\base\Application::basePath|basePath]].
2. Register the [[yii\base\Application::errorHandler|error handler]].
3. Initialize application properties using the given application configuration.
4. [[yii\base\Application::init()|init()]] is called which in turn calls [[yii\base\Application::bootstrap()|bootstrap()]] to run bootstrapping components.
    - Include the extension manifest file `vendor/yiisoft/extensions.php`.
    - Create and run bootstrap components declared by extensions.
    - Create and run application components and/or modules that are declared in the application's bootstrap property.

Because the bootstrapping work has to be done before handling *every* request, it is very important to keep this process light and optimize it as much as possible.

Try not to register too many bootstrapping components. A bootstrapping component is needed only if it wants to participate the whole life cycle of requesting handling. For example, if a module needs to register additional URL parsing rules, it should be listed in the bootstrap property so that the new URL rules can take effect before they are used to resolve requests.

In production mode, enable a bytecode cache, such as PHP OPcache or APC, to minimize the time needed for including and parsing PHP files.

Some large applications have very complex application configurations which are divided into many smaller configuration files. If this is the case, consider caching the whole configuration array and loading it directly from cache before creating the application instance in the entry script.

Routing and URL Creation

When a Yii application starts processing a requested URL, the first step it takes is to parse the URL into a route. The route is then used to instantiate the corresponding controller action to handle the request. This whole process is called *routing*.

The reverse process of routing is called *URL creation*, which creates a URL from a given route and the associated query parameters. When the created URL is later requested, the routing process can resolve it back into the original route and query parameters.

The central piece responsible for routing and URL creation is the [[yii\web\UrlManager|URL manager]], which is registered as the `urlManager` application component. The [[yii\web\UrlManager|URL manager]] provides the [[yii\web\UrlManager::parseRequest()|parseRequest()]] method to parse an incoming request into a route and the associated query parameters and the [[yii\web\UrlManager::createUrl()|createUrl()]] method to create a URL from a given route and its associated query parameters.

By configuring the `urlManager` component in the application configuration, you can let your application recognize arbitrary URL formats without modifying your existing application code. For example, you can use the following code to create a URL for the `post/view` action:

```
use yii\helpers\Url;

// Url::to() calls UrlManager::createUrl() to create a URL
$url = Url::to(['post/view', 'id' => 100]);
```

Depending on the `urlManager` configuration, the created URL may look like one of the following (or other format). And if the created URL is requested later, it will still be parsed back into the original route and query parameter value.

```
/index.php?r=post%2Fview&id=100
/index.php/post/100
/posts/100
```

URL Formats

The [[yii\web\UrlManager|URL manager]] supports two URL formats:

- the default URL format;

- the pretty URL format.

The default URL format uses a [[yii\web\UrlManager::$routeParam|query parameter]] named `r` to represent the route and normal query parameters to represent the query parameters associated with the route. For example, the URL `/index.php?r=post/view&id=100` represents the route `post/view` and the `id` query parameter `100`. The default URL format does not require any configuration of the [[yii\web\UrlManager|URL manager]] and works in any Web server setup.

The pretty URL format uses the extra path following the entry script name to represent the route and the associated query parameters. For example, the extra path in the URL `/index.php/post/100` is `/post/100` which may represent the route `post/view` and the `id` query parameter `100` with a proper [[yii\web\UrlManager::rules|URL rule]]. To use the pretty URL format, you will need to design a set of [[yii\web\UrlManager::rules|URL rules]] according to the actual requirement about how the URLs should look like.

You may switch between the two URL formats by toggling the [[yii\web\UrlManager::enablePrettyUrl|enablePrettyUrl]] property of the [[yii\web\UrlManager|URL manager]] without changing any other application code.

Routing

Routing involves two steps:

- the incoming request is parsed into a route and the associated query parameters;
- a controller action corresponding to the parsed route is created to handle the request.

When using the default URL format, parsing a request into a route is as simple as getting the value of a `GET` query parameter named `r`.

When using the pretty URL format, the [[yii\web\UrlManager|URL manager]] will examine the registered [[yii\web\UrlManager::rules|URL rules]] to find matching one that can resolve the request into a route. If such a rule cannot be found, a [[yii\web\NotFoundHttpException]] exception will be thrown.

Once the request is parsed into a route, it is time to create the controller action identified by the route. The route is broken down into multiple parts by the slashes in it. For example, `site/index` will be broken into `site` and `index`. Each part is an ID which may refer to a module, a controller or an action. Starting from the first part in the route, the application takes the following steps to create modules (if any), controller and action:

1. Set the application as the current module.
2. Check if the [[yii\base\Module::controllerMap|controller map]] of the current module

contains the current ID. If so, a controller object will be created according to the controller configuration found in the map, and Step 5 will be taken to handle the rest part of the route.
3. Check if the ID refers to a module listed in the [[yii\base\Module::modules|modules]] property of the current module. If so, a module is created according to the configuration found in the module list, and Step 2 will be taken to handle the next part of the route under the context of the newly created module.
4. Treat the ID as a controller ID and create a controller object. Do the next step with the rest part of the route.
5. The controller looks for the current ID in its [[yii\base\Controller::actions()|action map]]. If found, it creates an action according to the configuration found in the map. Otherwise, the controller will attempt to create an inline action which is defined by an action method corresponding to the current action ID.

Among the above steps, if any error occurs, a [[yii\web\NotFoundHttpException]] will be thrown, indicating the failure of the routing process.

Default Route

When a request is parsed into an empty route, the so-called *default route* will be used, instead. By default, the default route is `site/index`, which refers to the `index` action of the `site` controller. You may customize it by configuring the [[yii\web\Application::defaultRoute|defaultRoute]] property of the application in the application configuration like the following:

```
[
    // ...
    'defaultRoute' => 'main/index',
];
```

Similar to the default route of the application, there is also a default route for modules, so for example if there is a `user` module and the request is parsed into the route `user` the module's [[yii\base\Module::defaultRoute|defaultRoute]] is used to determine the controller. By default the controller name is `default`. If no action is specified in [[yii\base\Module::defaultRoute|defaultRoute]], the [[yii\base\Controller::defaultAction|defaultAction]] property of the controller is used to determine the action. In this example, the full route would be `user/default/index`.

`catchAll` Route

Sometimes, you may want to put your Web application in maintenance mode temporarily and display the same informational page for all requests. There are many ways to accomplish this goal. But one of the simplest ways is to configure the [[yii\web\Application::catchAll]] property like the following in the application configuration:

```
[
    // ...
    'catchAll' => ['site/offline'],
];
```

With the above configuration, the `site/offline` action will be used to handle all incoming requests.

The `catchAll` property should take an array whose first element specifies a route, and the rest of the elements (name-value pairs) specify the parameters to be bound to the action.

> Info: The debug toolbar in development environment will not work when this property is enabled.

Creating URLs

Yii provides a helper method [[yii\helpers\Url::to()]] to create various kinds of URLs from given routes and their associated query parameters. For example,

```
use yii\helpers\Url;

// creates a URL to a route: /index.php?r=post%2Findex
echo Url::to(['post/index']);

// creates a URL to a route with parameters: /index.php?r=post%2Fview&id=100
echo Url::to(['post/view', 'id' => 100]);

// creates an anchored URL: /index.php?r=post%2Fview&id=100#content
echo Url::to(['post/view', 'id' => 100, '#' => 'content']);

// creates an absolute URL: http://www.example.com/index.php?r=post%2Findex
echo Url::to(['post/index'], true);

// creates an absolute URL using the https scheme: https://www.example.com/index.php?r
=post%2Findex
echo Url::to(['post/index'], 'https');
```

Note that in the above example, we assume the default URL format is being used. If the pretty URL format is enabled, the created URLs will be different, according to the [[yii\web\UrlManager::rules|URL rules]] in use.

The route passed to the [[yii\helpers\Url::to()]] method is context sensitive. It can be either a *relative* route or an *absolute* route which will be normalized according to the following rules:

- If the route is an empty string, the currently requested [[yii\web\Controller::route|route]] will be used;
- If the route contains no slashes at all, it is considered to be an action ID of the current controller and will be prepended with the [[\yii\web\Controller::uniqueId|uniqueId]] value of the current controller;
- If the route has no leading slash, it is considered to be a route relative to the current module and will be prepended with the [[\yii\base\Module::uniqueId|uniqueId]] value of the current module.

Starting from version 2.0.2, you may specify a route in terms of an alias. If this is the case, the alias will first be converted into the actual route which will then be turned into an absolute route according to the above rules.

For example, assume the current module is `admin` and the current controller is `post`,

```
use yii\helpers\Url;

// currently requested route: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['']);

// a relative route with action ID only: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['index']);

// a relative route: /index.php?r=admin%2Fpost%2Findex
echo Url::to(['post/index']);

// an absolute route: /index.php?r=post%2Findex
echo Url::to(['/post/index']);

// using an alias "@posts", which is defined as "/post/index": /index.php?r=post%2Find
ex
echo Url::to(['@posts']);
```

The [[yii\helpers\Url::to()]] method is implemented by calling the [[yii\web\UrlManager::createUrl()|createUrl()]] and [[yii\web\UrlManager::createAbsoluteUrl()|createAbsoluteUrl()]] methods of the [[yii\web\UrlManager|URL manager]]. In the next few subsections, we will explain how to configure the [[yii\web\UrlManager|URL manager]] to customize the format of the created URLs.

The [[yii\helpers\Url::to()]] method also supports creating URLs that are **not** related with particular routes. Instead of passing an array as its first parameter, you should pass a string in this case. For example,

```
use yii\helpers\Url;

// currently requested URL: /index.php?r=admin%2Fpost%2Findex
echo Url::to();

// an aliased URL: http://example.com
Yii::setAlias('@example', 'http://example.com/');
echo Url::to('@example');

// an absolute URL: http://example.com/images/logo.gif
echo Url::to('/images/logo.gif', true);
```

Besides the `to()` method, the [[yii\helpers\Url]] helper class also provides several other convenient URL creation methods. For example,

```
use yii\helpers\Url;

// home page URL: /index.php?r=site%2Findex
echo Url::home();

// the base URL, useful if the application is deployed in a sub-folder of the Web root
echo Url::base();

// the canonical URL of the currently requested URL
// see https://en.wikipedia.org/wiki/Canonical_link_element
echo Url::canonical();

// remember the currently requested URL and retrieve it back in later requests
Url::remember();
echo Url::previous();
```

Using Pretty URLs

To use pretty URLs, configure the `urlManager` component in the application configuration like the following:

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
            'enableStrictParsing' => false,
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

The [[yii\web\UrlManager::enablePrettyUrl|enablePrettyUrl]] property is mandatory as it toggles the pretty URL format. The rest of the properties are optional. However, their configuration shown above is most commonly used.

- [[yii\web\UrlManager::showScriptName|showScriptName]]: this property determines whether the entry script should be included in the created URLs. For example, instead of creating a URL `/index.php/post/100`, by setting this property to be `false`, a URL `/post/100` will be generated.
- [[yii\web\UrlManager::enableStrictParsing|enableStrictParsing]]: this property determines whether to enable strict request parsing. If strict parsing is enabled, the incoming requested URL must match at least one of the [[yii\web\UrlManager::rules|rules]] in order to be treated as a valid request, otherwise a [[yii\web\NotFoundHttpException]] will be thrown. If strict parsing is disabled, when none of the [[yii\web\UrlManager::rules|rules]] matches the requested URL, the path info part of the URL will be treated as the requested route.
- [[yii\web\UrlManager::rules|rules]]: this property contains a list of rules specifying how to parse and create URLs. It is the main property that you should work with in order to create URLs whose format satisfies your particular application requirement.

> Note: In order to hide the entry script name in the created URLs, besides setting [[yii\web\UrlManager::showScriptName|showScriptName]] to be `false`, you may also need to configure your Web server so that it can correctly identify which PHP script should be executed when a requested URL does not explicitly specify one. If you are using Apache or nginx Web server, you may refer to the recommended configuration as described in the Installation section.

URL Rules

A URL rule is a class implementing the [[yii\web\UrlRuleInterface]], usually [[yii\web\UrlRule]]. Each URL rule consists of a pattern used for matching the path info part of URLs, a route, and a few query parameters. A URL rule can be used to parse a request if its pattern matches the requested URL. A URL rule can be used to create a URL if its route and query parameter names match those that are given.

When the pretty URL format is enabled, the [[yii\web\UrlManager|URL manager]] uses the URL rules declared in its [[yii\web\UrlManager::rules|rules]] property to parse incoming requests and create URLs. In particular, to parse an incoming request, the [[yii\web\UrlManager|URL manager]] examines the rules in the order they are declared and looks for the *first* rule that matches the requested URL. The matching rule is then used to parse the URL into a route and its associated parameters. Similarly, to create a URL, the [[yii\web\UrlManager|URL manager]] looks for the first rule that matches the given route and parameters and uses that to create a URL.

You can configure [[yii\web\UrlManager::rules]] as an array with keys being the [[yii\web\UrlRule::$pattern|patterns]] and values the corresponding [[yii\web\UrlRule::$route|routes]]. Each pattern-route pair constructs a URL rule. For example, the following [[yii\web\UrlManager::rules|rules]] configuration declares two URL rules. The first rule matches a URL `posts` and maps it into the route `post/index`. The second rule matches a URL matching the regular expression `post/(\d+)` and maps it into the route `post/view` and defines a query parameter named `id`.

```
'rules' => [
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

> Info: The pattern in a rule is used to match the path info part of a URL. For example, the path info of `/index.php/post/100?source=ad` is `post/100` (the leading and ending slashes are ignored) which matches the pattern `post/(\d+)`.

Besides declaring URL rules as pattern-route pairs, you may also declare them as configuration arrays. Each configuration array is used to configure a single URL rule object. This is often needed when you want to configure other properties of a URL rule. For example,

```
'rules' => [
    // ...other url rules...
    [
        'pattern' => 'posts',
        'route' => 'post/index',
        'suffix' => '.json',
    ],
]
```

By default if you do not specify the `class` option for a rule configuration, it will take the default class [[yii\web\UrlRule]], which is the default value defined in [[yii\web\UrlManager::$ruleConfig]].

Named Parameters

A URL rule can be associated with named query parameters which are specified in the pattern in the format of `<ParamName:RegExp>`, where `ParamName` specifies the parameter name and `RegExp` is an optional regular expression used to match parameter values. If `RegExp` is not specified, it means the parameter value should be a string without any slash.

> Note: You can only use regular expressions inside of parameters. The rest of a pattern is considered plain text.

When a rule is used to parse a URL, it will fill the associated parameters with values matching the corresponding parts of the URL, and these parameters will be made available in `$_GET` later by the `request` application component. When the rule is used to create a URL, it will take the values of the provided parameters and insert them at the places where the parameters are declared.

Let's use some examples to illustrate how named parameters work. Assume we have declared the following three URL rules:

```
'rules' => [
    'posts/<year:\d{4}>/<category>' => 'post/index',
    'posts' => 'post/index',
    'post/<id:\d+>' => 'post/view',
]
```

When the rules are used to parse URLs:

- `/index.php/posts` is parsed into the route `post/index` using the second rule;
- `/index.php/posts/2014/php` is parsed into the route `post/index`, the `year` parameter whose value is 2014 and the `category` parameter whose value is `php` using the first rule;

- `/index.php/post/100` is parsed into the route `post/view` and the `id` parameter whose value is 100 using the third rule;
- `/index.php/posts/php` will cause a [[yii\web\NotFoundHttpException]] when [[yii\web\UrlManager::enableStrictParsing]] is `true`, because it matches none of the patterns. If [[yii\web\UrlManager::enableStrictParsing]] is `false` (the default value), the path info part `posts/php` will be returned as the route. This will either execute the corresponding action if it exists or throw a [[yii\web\NotFoundHttpException]] otherwise.

And when the rules are used to create URLs:

- `Url::to(['post/index'])` creates `/index.php/posts` using the second rule;
- `Url::to(['post/index', 'year' => 2014, 'category' => 'php'])` creates `/index.php/posts/2014/php` using the first rule;
- `Url::to(['post/view', 'id' => 100])` creates `/index.php/post/100` using the third rule;
- `Url::to(['post/view', 'id' => 100, 'source' => 'ad'])` creates `/index.php/post/100?source=ad` using the third rule. Because the `source` parameter is not specified in the rule, it is appended as a query parameter in the created URL.
- `Url::to(['post/index', 'category' => 'php'])` creates `/index.php/post/index?category=php` using none of the rules. Note that since none of the rules applies, the URL is created by simply appending the route as the path info and all parameters as the query string part.

Parameterizing Routes

You can embed parameter names in the route of a URL rule. This allows a URL rule to be used for matching multiple routes. For example, the following rules embed `controller` and `action` parameters in the routes.

```
'rules' => [
    '<controller:(post|comment)>/create' => '<controller>/create',
    '<controller:(post|comment)>/<id:\d+>/<action:(update|delete)>' => '<controller>/<
action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/view',
    '<controller:(post|comment)>s' => '<controller>/index',
]
```

To parse a URL `/index.php/comment/100/update`, the second rule will apply, which sets the `controller` parameter to be `comment` and `action` parameter to be `update`. The route `<controller>/<action>` is thus resolved as `comment/update`.

Similarly, to create a URL for the route `comment/index`, the last rule will apply, which creates a URL `/index.php/comments`.

> Info: By parameterizing routes, it is possible to greatly reduce the number of URL rules, which can significantly improve the performance of [[yii\web\UrlManager|URL manager]].

Default Parameter Values

By default, all parameters declared in a rule are required. If a requested URL does not contain a particular parameter, or if a URL is being created without a particular parameter, the rule will not apply. To make some of the parameters optional, you can configure the [[yii\web\UrlRule::defaults|defaults]] property of a rule. Parameters listed in this property are optional and will take the specified values when they are not provided.

In the following rule declaration, the `page` and `tag` parameters are both optional and will take the value of 1 and empty string, respectively, when they are not provided.

```
'rules' => [
    // ...other rules...
    [
        'pattern' => 'posts/<page:\d+>/<tag>',
        'route' => 'post/index',
        'defaults' => ['page' => 1, 'tag' => ''],
    ],
]
```

The above rule can be used to parse or create any of the following URLs:

- `/index.php/posts`: `page` is 1, `tag` is ''.
- `/index.php/posts/2`: `page` is 2, `tag` is ''.
- `/index.php/posts/2/news`: `page` is 2, `tag` is `'news'`.
- `/index.php/posts/news`: `page` is 1, `tag` is `'news'`.

Without using optional parameters, you would have to create 4 rules to achieve the same result.

Rules with Server Names

It is possible to include Web server names in the patterns of URL rules. This is mainly useful when your application should behave differently for different Web server names. For example, the following rules will parse the URL `http://admin.example.com/login` into the route `admin/user/login` and `http://www.example.com/login` into `site/login`.

```
'rules' => [
    'http://admin.example.com/login' => 'admin/user/login',
    'http://www.example.com/login' => 'site/login',
]
```

You can also embed parameters in the server names to extract dynamic information from them. For example, the following rule will parse the URL `http://en.example.com/posts` into the route `post/index` and the parameter `language=en`.

```
'rules' => [
    'http://<language:\w+>.example.com/posts' => 'post/index',
]
```

Since version 2.0.11, you may also use protocol relative patterns that work for both, `http` and `https`. The syntax is the same as above but skipping the `http:` part, e.g.: `'//www.example.com/login' => 'site/login'`.

> Note: Rules with server names should **not** include the subfolder of the entry script in their patterns. For example, if the applications entry script is at `http://www.example.com/sandbox/blog/index.php`, then you should use the pattern `http://www.example.com/posts` instead of `http://www.example.com/sandbox/blog/posts`. This will allow your application to be deployed under any directory without the need to change your url rules. Yii will automatically detect the base url of the application.

URL Suffixes

You may want to add suffixes to the URLs for various purposes. For example, you may add `.html` to the URLs so that they look like URLs for static HTML pages; you may also add `.json` to the URLs to indicate the expected content type of the response. You can achieve this goal by configuring the [[yii\web\UrlManager::suffix]] property like the following in the application configuration:

```
[
    // ...
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            // ...
            'suffix' => '.html',
            'rules' => [
                // ...
            ],
        ],
    ],
]
```

The above configuration will allow the [[yii\web\UrlManager|URL manager]] to recognize requested URLs and also create URLs with `.html` as their suffix.

> Tip: You may set `/` as the URL suffix so that the URLs all end with a slash.
>
> Note: When you configure a URL suffix, if a requested URL does not have the suffix, it will be considered as an unrecognized URL. This is a recommended practice for SEO (search engine optimization) to avoid duplicate content on different URLs.

Sometimes you may want to use different suffixes for different URLs. This can be achieved by configuring the [[yii\web\UrlRule::suffix|suffix]] property of individual URL rules. When a URL rule has this property set, it will override the suffix setting at the [[yii\web\UrlManager|URL manager]] level. For example, the following configuration contains a customized URL rule which uses `.json` as its suffix instead of the global `.html` suffix.

```
[
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            // ...
            'suffix' => '.html',
            'rules' => [
                // ...
                [
                    'pattern' => 'posts',
                    'route' => 'post/index',
                    'suffix' => '.json',
                ],
            ],
        ],
    ],
]
```

HTTP Methods

When implementing RESTful APIs, it is commonly needed that the same URL be parsed into different routes according to the HTTP methods being used. This can be easily achieved by prefixing the supported HTTP methods to the patterns of the rules. If a rule supports multiple HTTP methods, separate the method names with commas. For example, the following rules have the same pattern `post/<id:\d+>` with different HTTP method support. A request for `PUT post/100` will be parsed into `post/update`, while a request for `GET post/100` will be parsed into `post/view`.

```
'rules' => [
    'PUT,POST post/<id:\d+>' => 'post/update',
    'DELETE post/<id:\d+>' => 'post/delete',
    'post/<id:\d+>' => 'post/view',
]
```

> Note: If a URL rule contains HTTP method(s) in its pattern, the rule will only be used for parsing purpose unless `GET` is among the specified verbs. It will be skipped when the [[yii\web\UrlManager|URL manager]] is called to create URLs.
>
> Tip: To simplify the routing of RESTful APIs, Yii provides a special URL rule class [[yii\rest\UrlRule]] which is very efficient and supports some fancy features such as automatic pluralization of controller IDs. For more details, please refer to the Routing section in the RESTful APIs chapter.

Adding Rules Dynamically

URL rules can be dynamically added to the [[yii\web\UrlManager|URL manager]]. This is often needed by redistributable modules which want to manage their own URL rules. In order for the dynamically added rules to take effect during the routing process, you should add them during the bootstrapping stage of the application. For modules, this means they should implement [[yii\base\BootstrapInterface]] and add the rules in the [[yii\base\BootstrapInterface::bootstrap()|bootstrap()]] method like the following:

```
public function bootstrap($app)
{
    $app->getUrlManager()->addRules([
        // rule declarations here
    ], false);
}
```

Note that you should also list these modules in [[yii\web\Application::bootstrap]] so that they can participate the bootstrapping process.

Creating Rule Classes

Despite the fact that the default [[yii\web\UrlRule]] class is flexible enough for the majority of projects, there are situations when you have to create your own rule classes. For example, in a car dealer Web site, you may want to support the URL format like `/Manufacturer/Model`, where both `Manufacturer` and `Model` must match some data stored in a database table. The default rule class will not work here because it relies on statically declared patterns.

We can create the following URL rule class to solve this problem.

```
<?php

namespace app\components;

use yii\web\UrlRuleInterface;
use yii\base\Object;

class CarUrlRule extends Object implements UrlRuleInterface
{
    public function createUrl($manager, $route, $params)
    {
        if ($route === 'car/index') {
            if (isset($params['manufacturer'], $params['model'])) {
                return $params['manufacturer'] . '/' . $params['model'];
            } elseif (isset($params['manufacturer'])) {
                return $params['manufacturer'];
            }
        }
        return false; // this rule does not apply
    }

    public function parseRequest($manager, $request)
    {
        $pathInfo = $request->getPathInfo();
        if (preg_match('%^(\w+)(/(\w+))?$%', $pathInfo, $matches)) {
            // check $matches[1] and $matches[3] to see
            // if they match a manufacturer and a model in the database.
            // If so, set $params['manufacturer'] and/or $params['model']
            // and return ['car/index', $params]
        }
        return false; // this rule does not apply
    }
}
```

And use the new rule class in the [[yii\web\UrlManager::rules]] configuration:

```
'rules' => [
    // ...other rules...
    [
        'class' => 'app\components\CarUrlRule',
        // ...configure other properties...
    ],
]
```

URL normalization

Since version 2.0.10 [[yii\web\UrlManager|UrlManager]] can be configured to use [[yii\web\UrlNormalizer|UrlNormalizer]] for dealing with variations of the same URL, for example with and without a trailing slash. Because technically `http://example.com/path` and `http://example.com/path/` are different URLs, serving the same content for both of them can degrade SEO ranking. By default normalizer collapses consecutive slashes, adds or removes trailing slashes depending on whether the suffix has a trailing slash or not, and redirects to the normalized version of the URL using permanent redirection. The normalizer can be configured globally for the URL manager or individually for each rule - by default each rule will use the normalizer from URL manager. You can set [[yii\web\UrlRule::$normalizer|UrlRule::$normalizer]] to `false` to disable normalization for particular URL rule.

The following shows an example configuration for the [[yii\web\UrlNormalizer|UrlNormalizer]]:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'showScriptName' => false,
    'enableStrictParsing' => true,
    'suffix' => '.html',
    'normalizer' => [
        'class' => 'yii\web\UrlNormalizer',
        // use temporary redirection instead of permanent for debugging
        'action' => UrlNormalizer::ACTION_REDIRECT_TEMPORARY,
    ],
    'rules' => [
        // ...other rules...
        [
            'pattern' => 'posts',
            'route' => 'post/index',
            'suffix' => '/',
            'normalizer' => false, // disable normalizer for this rule
        ],
        [
            'pattern' => 'tags',
            'route' => 'tag/index',
            'normalizer' => [
                // do not collapse consecutive slashes for this rule
                'collapseSlashes' => false,
            ],
        ],
    ],
]
```

> Note: by default [[yii\web\UrlManager::$normalizer|UrlManager::$normalizer]] is disabled. You need to explicitly configure it in order to enable URL normalization.

Performance Considerations

When developing a complex Web application, it is important to optimize URL rules so that it takes less time to parse requests and create URLs.

By using parameterized routes, you may reduce the number of URL rules, which can significantly improve performance.

When parsing or creating URLs, [[yii\web\UrlManager|URL manager]] examines URL rules in the order they are declared. Therefore, you may consider adjusting the order of the URL rules so that more specific and/or more commonly used rules are placed before less used ones.

If some URL rules share the same prefix in their patterns or routes, you may consider using [[yii\web\GroupUrlRule]] so that they can be more efficiently examined by [[yii\web\UrlManager|URL manager]] as a group. This is often the case when your application is composed by modules, each having its own set of URL rules with module ID as their common prefixes.

Requests

Requests made to an application are represented in terms of [[yii\web\Request]] objects which provide information such as request parameters, HTTP headers, cookies, etc. For a given request, you can get access to the corresponding request object via the `request` application component which is an instance of [[yii\web\Request]], by default. In this section, we will describe how you can make use of this component in your applications.

Request Parameters

To get request parameters, you can call [[yii\web\Request::get()|get()]] and [[yii\web\Request::post()|post()]] methods of the `request` component. They return the values of `$_GET` and `$_POST`, respectively. For example,

```
$request = Yii::$app->request;

$get = $request->get();
// equivalent to: $get = $_GET;

$id = $request->get('id');
// equivalent to: $id = isset($_GET['id']) ? $_GET['id'] : null;

$id = $request->get('id', 1);
// equivalent to: $id = isset($_GET['id']) ? $_GET['id'] : 1;

$post = $request->post();
// equivalent to: $post = $_POST;

$name = $request->post('name');
// equivalent to: $name = isset($_POST['name']) ? $_POST['name'] : null;

$name = $request->post('name', '');
// equivalent to: $name = isset($_POST['name']) ? $_POST['name'] : '';
```

> Info: Instead of directly accessing `$_GET` and `$_POST` to retrieve the request parameters, it is recommended that you get them via the `request` component as shown above. This will make writing tests easier because you can create a mock request component with faked request data.

When implementing RESTful APIs, you often need to retrieve parameters that are submitted via PUT, PATCH or other request methods. You can get these parameters by calling the [[yii\web\Request::getBodyParam()]] methods. For example,

```
$request = Yii::$app->request;

// returns all parameters
$params = $request->bodyParams;

// returns the parameter "id"
$param = $request->getBodyParam('id');
```

> Info: Unlike `GET` parameters, parameters submitted via `POST`, `PUT`, `PATCH` etc. are sent in the request body. The `request` component will parse these parameters when you access them through the methods described above. You can customize the way how these parameters are parsed by configuring the [[yii\web\Request::parsers]] property.

Request Methods

You can get the HTTP method used by the current request via the expression `Yii::$app->request->method`. A whole set of boolean properties is also provided for you to check if the current method is of certain type. For example,

```
$request = Yii::$app->request;

if ($request->isAjax) { /* the request is an AJAX request */ }
if ($request->isGet)  { /* the request method is GET */ }
if ($request->isPost) { /* the request method is POST */ }
if ($request->isPut)  { /* the request method is PUT */ }
```

Request URLs

The `request` component provides many ways of inspecting the currently requested URL.

Assuming the URL being requested is `http://example.com/admin/index.php/product?id=100`, you can get various parts of this URL as summarized in the following:

- [[yii\web\Request::url|url]]: returns `/admin/index.php/product?id=100`, which is the URL without the host info part.
- [[yii\web\Request::absoluteUrl|absoluteUrl]]: returns `http://example.com/admin/index.php/product?id=100`, which is the whole URL including the host info part.
- [[yii\web\Request::hostInfo|hostInfo]]: returns `http://example.com`, which is the host info part of the URL.

- [[yii\web\Request::pathInfo|pathInfo]]: returns `/product`, which is the part after the entry script and before the question mark (query string).
- [[yii\web\Request::queryString|queryString]]: returns `id=100`, which is the part after the question mark.
- [[yii\web\Request::baseUrl|baseUrl]]: returns `/admin`, which is the part after the host info and before the entry script name.
- [[yii\web\Request::scriptUrl|scriptUrl]]: returns `/admin/index.php`, which is the URL without path info and query string.
- [[yii\web\Request::serverName|serverName]]: returns `example.com`, which is the host name in the URL.
- [[yii\web\Request::serverPort|serverPort]]: returns 80, which is the port used by the Web server.

HTTP Headers

You can get the HTTP header information through the [[yii\web\HeaderCollection|header collection]] returned by the [[yii\web\Request::headers]] property. For example,

```
// $headers is an object of yii\web\HeaderCollection
$headers = Yii::$app->request->headers;

// returns the Accept header value
$accept = $headers->get('Accept');

if ($headers->has('User-Agent')) { /* there is User-Agent header */ }
```

The `request` component also provides support for quickly accessing some commonly used headers, including:

- [[yii\web\Request::userAgent|userAgent]]: returns the value of the `User-Agent` header.
- [[yii\web\Request::contentType|contentType]]: returns the value of the `Content-Type` header which indicates the MIME type of the data in the request body.
- [[yii\web\Request::acceptableContentTypes|acceptableContentTypes]]: returns the content MIME types acceptable by users. The returned types are ordered by their quality score. Types with the highest scores will be returned first.
- [[yii\web\Request::acceptableLanguages|acceptableLanguages]]: returns the languages acceptable by users. The returned languages are ordered by their preference level. The first element represents the most preferred language.

If your application supports multiple languages and you want to display pages in the language that is the most preferred by the end user, you may use the language negotiation method [[yii\web\Request::getPreferredLanguage()]]. This method takes a list of languages

supported by your application, compares them with [[yii\web\Request::acceptableLanguages|acceptableLanguages]], and returns the most appropriate language.

> Tip: You may also use the [[yii\filters\ContentNegotiator|ContentNegotiator]] filter to dynamically determine what content type and language should be used in the response. The filter implements the content negotiation on top of the properties and methods described above.

Client Information

You can get the host name and IP address of the client machine through [[yii\web\Request::userHost|userHost]] and [[yii\web\Request::userIP|userIP]], respectively. For example,

```
$userHost = Yii::$app->request->userHost;
$userIP = Yii::$app->request->userIP;
```

Responses

When an application finishes handling a request, it generates a [[yii\web\Response|response]] object and sends it to the end user. The response object contains information such as the HTTP status code, HTTP headers and body. The ultimate goal of Web application development is essentially to build such response objects upon various requests.

In most cases you should mainly deal with the `response` application component which is an instance of [[yii\web\Response]], by default. However, Yii also allows you to create your own response objects and send them to end users as we will explain in the following.

In this section, we will describe how to compose and send responses to end users.

Status Code

One of the first things you would do when building a response is to state whether the request is successfully handled. This is done by setting the [[yii\web\Response::statusCode]] property which can take one of the valid HTTP status codes. For example, to indicate the request is successfully handled, you may set the status code to be 200, like the following:

```
Yii::$app->response->statusCode = 200;
```

However, in most cases you do not need to explicitly set the status code. This is because the default value of [[yii\web\Response::statusCode]] is 200. And if you want to indicate the request is unsuccessful, you may throw an appropriate HTTP exception like the following:

```
throw new \yii\web\NotFoundHttpException;
```

When the error handler catches an exception, it will extract the status code from the exception and assign it to the response. For the [[yii\web\NotFoundHttpException]] above, it is associated with the HTTP status 404. The following HTTP exceptions are predefined in Yii:

- [[yii\web\BadRequestHttpException]]: status code 400.
- [[yii\web\ConflictHttpException]]: status code 409.
- [[yii\web\ForbiddenHttpException]]: status code 403.
- [[yii\web\GoneHttpException]]: status code 410.
- [[yii\web\MethodNotAllowedHttpException]]: status code 405.

- [[yii\web\NotAcceptableHttpException]]: status code 406.
- [[yii\web\NotFoundHttpException]]: status code 404.
- [[yii\web\ServerErrorHttpException]]: status code 500.
- [[yii\web\TooManyRequestsHttpException]]: status code 429.
- [[yii\web\UnauthorizedHttpException]]: status code 401.
- [[yii\web\UnsupportedMediaTypeHttpException]]: status code 415.

If the exception that you want to throw is not among the above list, you may create one by extending from [[yii\web\HttpException]], or directly throw it with a status code, for example,

```
throw new \yii\web\HttpException(402);
```

HTTP Headers

You can send HTTP headers by manipulating the [[yii\web\Response::headers|header collection]] in the `response` component. For example,

```
$headers = Yii::$app->response->headers;

// add a Pragma header. Existing Pragma headers will NOT be overwritten.
$headers->add('Pragma', 'no-cache');

// set a Pragma header. Any existing Pragma headers will be discarded.
$headers->set('Pragma', 'no-cache');

// remove Pragma header(s) and return the removed Pragma header values in an array
$values = $headers->remove('Pragma');
```

> Info: Header names are case insensitive. And the newly registered headers are not sent to the user until the [[yii\web\Response::send()]] method is called.

Response Body

Most responses should have a body which gives the content that you want to show to end users.

If you already have a formatted body string, you may assign it to the [[yii\web\Response::content]] property of the response. For example,

```
Yii::$app->response->content = 'hello world!';
```

If your data needs to be formatted before sending it to end users, you should set both of the [[yii\web\Response::format|format]] and [[yii\web\Response::data|data]] properties. The [[yii\web\Response::format|format]] property specifies in which format the [[yii\web\Response::data|data]] should be formatted. For example,

```
$response = Yii::$app->response;
$response->format = \yii\web\Response::FORMAT_JSON;
$response->data = ['message' => 'hello world'];
```

Yii supports the following formats out of the box, each implemented by a [[yii\web\ResponseFormatterInterface|formatter]] class. You can customize these formatters or add new ones by configuring the [[yii\web\Response::formatters]] property.

- [[yii\web\Response::FORMAT_HTML|HTML]]: implemented by [[yii\web\HtmlResponseFormatter]].
- [[yii\web\Response::FORMAT_XML|XML]]: implemented by [[yii\web\XmlResponseFormatter]].
- [[yii\web\Response::FORMAT_JSON|JSON]]: implemented by [[yii\web\JsonResponseFormatter]].
- [[yii\web\Response::FORMAT_JSONP|JSONP]]: implemented by [[yii\web\JsonResponseFormatter]].
- [[yii\web\Response::FORMAT_RAW|RAW]]: use this format if you want to send the response directly without applying any formatting.

While the response body can be set explicitly as shown above, in most cases you may set it implicitly by the return value of action methods. A common use case is like the following:

```
public function actionIndex()
{
    return $this->render('index');
}
```

The `index` action above returns the rendering result of the `index` view. The return value will be taken by the `response` component, formatted and then sent to end users.

Because by default the response format is [[yii\web\Response::FORMAT_HTML|HTML]], you should only return a string in an action method. If you want to use a different response format, you should set it first before returning the data. For example,

```
public function actionInfo()
{
    \Yii::$app->response->format = \yii\web\Response::FORMAT_JSON;
    return [
        'message' => 'hello world',
        'code' => 100,
    ];
}
```

As aforementioned, besides using the default `response` application component, you can also create your own response objects and send them to end users. You can do so by returning such object in an action method, like the following,

```
public function actionInfo()
{
    return \Yii::createObject([
        'class' => 'yii\web\Response',
        'format' => \yii\web\Response::FORMAT_JSON,
        'data' => [
            'message' => 'hello world',
            'code' => 100,
        ],
    ]);
}
```

> Note: If you are creating your own response objects, you will not be able to take advantage of the configurations that you set for the `response` component in the application configuration. You can, however, use dependency injection to apply a common configuration to your new response objects.

Browser Redirection

Browser redirection relies on sending a `Location` HTTP header. Because this feature is commonly used, Yii provides some special support for it.

You can redirect the user browser to a URL by calling the [[yii\web\Response::redirect()]] method. The method sets the appropriate `Location` header with the given URL and returns the response object itself. In an action method, you can call its shortcut version [[yii\web\Controller::redirect()]]. For example,

```
public function actionOld()
{
    return $this->redirect('http://example.com/new', 301);
}
```

In the above code, the action method returns the result of the `redirect()` method. As explained before, the response object returned by an action method will be used as the response sending to end users.

In places other than an action method, you should call [[yii\web\Response::redirect()]] directly followed by a chained call to the [[yii\web\Response::send()]] method to ensure no extra content will be appended to the response.

```
\Yii::$app->response->redirect('http://example.com/new', 301)->send();
```

> Info: By default, the [[yii\web\Response::redirect()]] method sets the response status code to be 302 which instructs the browser that the resource being requested is *temporarily* located in a different URI. You can pass in a status code 301 to tell the browser that the resource has been *permanently* relocated.

When the current request is an AJAX request, sending a `Location` header will not automatically cause the browser to redirect. To solve this problem, the [[yii\web\Response::redirect()]] method sets an `X-Redirect` header with the redirection URL as its value. On the client-side, you may write JavaScript code to read this header value and redirect the browser accordingly.

> Info: Yii comes with a `yii.js` JavaScript file which provides a set of commonly used JavaScript utilities, including browser redirection based on the `X-Redirect` header. Therefore, if you are using this JavaScript file (by registering the [[yii\web\YiiAsset]] asset bundle), you do not need to write anything to support AJAX redirection. More information about `yii.js` can be found in the Client Scripts Section.

Sending Files

Like browser redirection, file sending is another feature that relies on specific HTTP headers. Yii provides a set of methods to support various file sending needs. They all have built-in support for the HTTP range header.

- [[yii\web\Response::sendFile()]]: sends an existing file to a client.
- [[yii\web\Response::sendContentAsFile()]]: sends a text string as a file to a client.
- [[yii\web\Response::sendStreamAsFile()]]: sends an existing file stream as a file to a client.

These methods have the same method signature with the response object as the return value. If the file to be sent is very big, you should consider using [[yii\web\Response::sendStreamAsFile()]] because it is more memory efficient. The following example shows how to send a file in a controller action:

```
public function actionDownload()
{
    return \Yii::$app->response->sendFile('path/to/file.txt');
}
```

If you are calling the file sending method in places other than an action method, you should also call the [[yii\web\Response::send()]] method afterwards to ensure no extra content will be appended to the response.

```
\Yii::$app->response->sendFile('path/to/file.txt')->send();
```

Some Web servers have a special file sending support called *X-Sendfile*. The idea is to redirect the request for a file to the Web server which will directly serve the file. As a result, the Web application can terminate earlier while the Web server is sending the file. To use this feature, you may call the [[yii\web\Response::xSendFile()]]. The following list summarizes how to enable the `X-Sendfile` feature for some popular Web servers:

- Apache: X-Sendfile
- Lighttpd v1.4: X-LIGHTTPD-send-file
- Lighttpd v1.5: X-Sendfile
- Nginx: X-Accel-Redirect
- Cherokee: X-Sendfile and X-Accel-Redirect

Sending Response

The content in a response is not sent to the user until the [[yii\web\Response::send()]] method is called. By default, this method will be called automatically at the end of [[yii\base\Application::run()]]. You can, however, explicitly call this method to force sending out the response immediately.

The [[yii\web\Response::send()]] method takes the following steps to send out a response:

1. Trigger the [[yii\web\Response::EVENT_BEFORE_SEND]] event.
2. Call [[yii\web\Response::prepare()]] to format [[yii\web\Response::data|response data]] into [[yii\web\Response::content|response content]].
3. Trigger the [[yii\web\Response::EVENT_AFTER_PREPARE]] event.
4. Call [[yii\web\Response::sendHeaders()]] to send out the registered HTTP headers.
5. Call [[yii\web\Response::sendContent()]] to send out the response body content.
6. Trigger the [[yii\web\Response::EVENT_AFTER_SEND]] event.

After the [[yii\web\Response::send()]] method is called once, any further call to this method will be ignored. This means once the response is sent out, you will not be able to append more content to it.

As you can see, the [[yii\web\Response::send()]] method triggers several useful events. By responding to these events, it is possible to adjust or decorate the response.

Sessions and Cookies

Sessions and cookies allow data to be persisted across multiple user requests. In plain PHP you may access them through the global variables `$_SESSION` and `$_COOKIE`, respectively. Yii encapsulates sessions and cookies as objects and thus allows you to access them in an object-oriented fashion with additional useful enhancements.

Sessions

Like requests and responses, you can get access to sessions via the `session` application component which is an instance of [[yii\web\Session]], by default.

Opening and Closing Sessions

To open and close a session, you can do the following:

```
$session = Yii::$app->session;

// check if a session is already open
if ($session->isActive) ...

// open a session
$session->open();

// close a session
$session->close();

// destroys all data registered to a session.
$session->destroy();
```

You can call [[yii\web\Session::open()|open()]] and [[yii\web\Session::close()|close()]] multiple times without causing errors; internally the methods will first check if the session is already open.

Accessing Session Data

To access the data stored in session, you can do the following:

```
$session = Yii::$app->session;

// get a session variable. The following usages are equivalent:
$language = $session->get('language');
$language = $session['language'];
$language = isset($_SESSION['language']) ? $_SESSION['language'] : null;

// set a session variable. The following usages are equivalent:
$session->set('language', 'en-US');
$session['language'] = 'en-US';
$_SESSION['language'] = 'en-US';

// remove a session variable. The following usages are equivalent:
$session->remove('language');
unset($session['language']);
unset($_SESSION['language']);

// check if a session variable exists. The following usages are equivalent:
if ($session->has('language')) ...
if (isset($session['language'])) ...
if (isset($_SESSION['language'])) ...

// traverse all session variables. The following usages are equivalent:
foreach ($session as $name => $value) ...
foreach ($_SESSION as $name => $value) ...
```

> Info: When you access session data through the `session` component, a session will be automatically opened if it has not been done so before. This is different from accessing session data through `$_SESSION`, which requires an explicit call of `session_start()`.

When working with session data that are arrays, the `session` component has a limitation which prevents you from directly modifying an array element. For example,

```
$session = Yii::$app->session;

// the following code will NOT work
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// the following code works:
$session['captcha'] = [
    'number' => 5,
    'lifetime' => 3600,
];

// the following code also works:
echo $session['captcha']['lifetime'];
```

You can use one of the following workarounds to solve this problem:

```
$session = Yii::$app->session;

// directly use $_SESSION (make sure Yii::$app->session->open() has been called)
$_SESSION['captcha']['number'] = 5;
$_SESSION['captcha']['lifetime'] = 3600;

// get the whole array first, modify it and then save it back
$captcha = $session['captcha'];
$captcha['number'] = 5;
$captcha['lifetime'] = 3600;
$session['captcha'] = $captcha;

// use ArrayObject instead of array
$session['captcha'] = new \ArrayObject;
...
$session['captcha']['number'] = 5;
$session['captcha']['lifetime'] = 3600;

// store array data by keys with a common prefix
$session['captcha.number'] = 5;
$session['captcha.lifetime'] = 3600;
```

For better performance and code readability, we recommend the last workaround. That is, instead of storing an array as a single session variable, you store each array element as a session variable which shares the same key prefix with other array elements.

Custom Session Storage

The default [[yii\web\Session]] class stores session data as files on the server. Yii also provides the following session classes implementing different session storage:

- [[yii\web\DbSession]]: stores session data in a database table.
- [[yii\web\CacheSession]]: stores session data in a cache with the help of a configured cache component.
- [[yii\redis\Session]]: stores session data using redis as the storage medium.
- [[yii\mongodb\Session]]: stores session data in a MongoDB.

All these session classes support the same set of API methods. As a result, you can switch to a different session storage class without the need to modify your application code that uses sessions.

> Note: If you want to access session data via `$_SESSION` while using custom session storage, you must make sure that the session has already been started by [[yii\web\Session::open()]]. This is because custom session storage handlers are registered within this method.

To learn how to configure and use these component classes, please refer to their API documentation. Below is an example showing how to configure [[yii\web\DbSession]] in the application configuration to use a database table for session storage:

```
return [
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',
            // 'db' => 'mydb',  // the application component ID of the DB connection.
Defaults to 'db'.
            // 'sessionTable' => 'my_session', // session table name. Defaults to 'ses
sion'.
        ],
    ],
];
```

You also need to create the following database table to store session data:

```
CREATE TABLE session
(
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

where 'BLOB' refers to the BLOB-type of your preferred DBMS. Below are the BLOB types that can be used for some popular DBMS:

- MySQL: LONGBLOB
- PostgreSQL: BYTEA
- MSSQL: BLOB

> Note: According to the php.ini setting of `session.hash_function`, you may need to adjust the length of the `id` column. For example, if `session.hash_function=sha256`, you should use a length 64 instead of 40.

Flash Data

Flash data is a special kind of session data which, once set in one request, will only be available during the next request and will be automatically deleted afterwards. Flash data is most commonly used to implement messages that should only be displayed to end users once, such as a confirmation message displayed after a user successfully submits a form.

You can set and access flash data through the `session` application component. For example,

```
$session = Yii::$app->session;

// Request #1
// set a flash message named as "postDeleted"
$session->setFlash('postDeleted', 'You have successfully deleted your post.');

// Request #2
// display the flash message named "postDeleted"
echo $session->getFlash('postDeleted');

// Request #3
// $result will be false since the flash message was automatically deleted
$result = $session->hasFlash('postDeleted');
```

Like regular session data, you can store arbitrary data as flash data.

When you call [[yii\web\Session::setFlash()]], it will overwrite any existing flash data that has the same name. To append new flash data to an existing message of the same name, you may call [[yii\web\Session::addFlash()]] instead. For example:

```
$session = Yii::$app->session;

// Request #1
// add a few flash messages under the name of "alerts"
$session->addFlash('alerts', 'You have successfully deleted your post.');
$session->addFlash('alerts', 'You have successfully added a new friend.');
$session->addFlash('alerts', 'You are promoted.');

// Request #2
// $alerts is an array of the flash messages under the name of "alerts"
$alerts = $session->getFlash('alerts');
```

> Note: Try not to use [[yii\web\Session::setFlash()]] together with [[yii\web\Session::addFlash()]] for flash data of the same name. This is because the latter method will automatically turn the flash data into an array so that it can append new flash data of the same name. As a result, when you call [[yii\web\Session::getFlash()]], you may find sometimes you are getting an array while sometimes you are getting a string, depending on the order of the invocation of these two methods.
>
> Tip: For displaying Flash messages you can use [[yii\bootstrap\Alert|bootstrap Alert]] widget in the following way:

```
echo Alert::widget([
   'options' => ['class' => 'alert-info'],
   'body' => Yii::$app->session->getFlash('postDeleted'),
]);
```

Cookies

Yii represents each cookie as an object of [[yii\web\Cookie]]. Both [[yii\web\Request]] and [[yii\web\Response]] maintain a collection of cookies via the property named `cookies`. The cookie collection in the former represents the cookies submitted in a request, while the cookie collection in the latter represents the cookies that are to be sent to the user.

The part of the application dealing with request and response directly is controller. Therefore, cookies should be read and sent in controller.

Reading Cookies

You can get the cookies in the current request using the following code:

```
// get the cookie collection (yii\web\CookieCollection) from the "request" component
$cookies = Yii::$app->request->cookies;

// get the "language" cookie value. If the cookie does not exist, return "en" as the d
efault value.
$language = $cookies->getValue('language', 'en');

// an alternative way of getting the "language" cookie value
if (($cookie = $cookies->get('language')) !== null) {
    $language = $cookie->value;
}

// you may also use $cookies like an array
if (isset($cookies['language'])) {
    $language = $cookies['language']->value;
}

// check if there is a "language" cookie
if ($cookies->has('language')) ...
if (isset($cookies['language'])) ...
```

Sending Cookies

You can send cookies to end users using the following code:

```
// get the cookie collection (yii\web\CookieCollection) from the "response" component
$cookies = Yii::$app->response->cookies;

// add a new cookie to the response to be sent
$cookies->add(new \yii\web\Cookie([
    'name' => 'language',
    'value' => 'zh-CN',
]));

// remove a cookie
$cookies->remove('language');
// equivalent to the following
unset($cookies['language']);
```

Besides the [[yii\web\Cookie::name|name]], [[yii\web\Cookie::value|value]] properties shown in the above examples, the [[yii\web\Cookie]] class also defines other properties to fully represent all available cookie information, such as [[yii\web\Cookie::domain|domain]], [[yii\web\Cookie::expire|expire]]. You may configure these properties as needed to prepare a cookie and then add it to the response's cookie collection.

> Note: For better security, the default value of [[yii\web\Cookie::httpOnly]] is set to `true`. This helps mitigate the risk of a client-side script accessing the protected cookie (if the browser supports it). You may read the httpOnly wiki article for more details.

Cookie Validation

When you are reading and sending cookies through the `request` and `response` components as shown in the last two subsections, you enjoy the added security of cookie validation which protects cookies from being modified on the client-side. This is achieved by signing each cookie with a hash string, which allows the application to tell if a cookie has been modified on the client-side. If so, the cookie will NOT be accessible through the [[yii\web\Request::cookies|cookie collection]] of the `request` component.

> Note: Cookie validation only protects cookie values from being modified. If a cookie fails the validation, you may still access it through `$_COOKIE`. This is because third-party libraries may manipulate cookies in their own way, which does not involve cookie validation.

Cookie validation is enabled by default. You can disable it by setting the [[yii\web\Request::enableCookieValidation]] property to be `false`, although we strongly recommend you do not do so.

> Note: Cookies that are directly read/sent via `$_COOKIE` and `setcookie()` will NOT be validated.

When using cookie validation, you must specify a [[yii\web\Request::cookieValidationKey]] that will be used to generate the aforementioned hash strings. You can do so by configuring the `request` component in the application configuration:

```
return [
    'components' => [
        'request' => [
            'cookieValidationKey' => 'fill in a secret key here',
        ],
    ],
];
```

> Info: [[yii\web\Request::cookieValidationKey|cookieValidationKey]] is critical to your application's security. It should only be known to people you trust. Do not store it in the version control system.

Handling Errors

Yii includes a built-in [[yii\web\ErrorHandler|error handler]] which makes error handling a much more pleasant experience than before. In particular, the Yii error handler does the following to improve error handling:

- All non-fatal PHP errors (e.g. warnings, notices) are converted into catchable exceptions.
- Exceptions and fatal PHP errors are displayed with detailed call stack information and source code lines in debug mode.
- Supports using a dedicated controller action to display errors.
- Supports different error response formats.

The [[yii\web\ErrorHandler|error handler]] is enabled by default. You may disable it by defining the constant `YII_ENABLE_ERROR_HANDLER` to be `false` in the entry script of your application.

Using Error Handler

The [[yii\web\ErrorHandler|error handler]] is registered as an application component named `errorHandler`. You may configure it in the application configuration like the following:

```
return [
    'components' => [
        'errorHandler' => [
            'maxSourceLines' => 20,
        ],
    ],
];
```

With the above configuration, the number of source code lines to be displayed in exception pages will be up to 20.

As aforementioned, the error handler turns all non-fatal PHP errors into catchable exceptions. This means you can use the following code to deal with PHP errors:

```
use Yii;
use yii\base\ErrorException;

try {
    10/0;
} catch (ErrorException $e) {
    Yii::warning("Division by zero.");
}

// execution continues...
```

If you want to show an error page telling the user that his request is invalid or unexpected, you may simply throw an [[yii\web\HttpException|HTTP exception]], such as [[yii\web\NotFoundHttpException]]. The error handler will correctly set the HTTP status code of the response and use an appropriate error view to display the error message.

```
use yii\web\NotFoundHttpException;

throw new NotFoundHttpException();
```

Customizing Error Display

The [[yii\web\ErrorHandler|error handler]] adjusts the error display according to the value of the constant `YII_DEBUG`. When `YII_DEBUG` is `true` (meaning in debug mode), the error handler will display exceptions with detailed call stack information and source code lines to help easier debugging. And when `YII_DEBUG` is `false`, only the error message will be displayed to prevent revealing sensitive information about the application.

> Info: If an exception is a descendant of [[yii\base\UserException]], no call stack will be displayed regardless the value of `YII_DEBUG`. This is because such exceptions are considered to be caused by user mistakes and the developers do not need to fix anything.

By default, the [[yii\web\ErrorHandler|error handler]] displays errors using two views:

- `@yii/views/errorHandler/error.php`: used when errors should be displayed WITHOUT call stack information. When `YII_DEBUG` is `false`, this is the only error view to be displayed.
- `@yii/views/errorHandler/exception.php`: used when errors should be displayed WITH call stack information.

You can configure the [[yii\web\ErrorHandler::errorView|errorView]] and [[yii\web\ErrorHandler::exceptionView|exceptionView]] properties of the error handler to use your own views to customize the error display.

Using Error Actions

A better way of customizing the error display is to use dedicated error actions. To do so, first configure the [[yii\web\ErrorHandler::errorAction|errorAction]] property of the `errorHandler` component like the following:

```
return [
    'components' => [
        'errorHandler' => [
            'errorAction' => 'site/error',
        ],
    ]
];
```

The [[yii\web\ErrorHandler::errorAction|errorAction]] property takes a route to an action. The above configuration states that when an error needs to be displayed without call stack information, the `site/error` action should be executed.

You can create the `site/error` action as follows,

```
namespace app\controllers;

use Yii;
use yii\web\Controller;

class SiteController extends Controller
{
    public function actions()
    {
        return [
            'error' => [
                'class' => 'yii\web\ErrorAction',
            ],
        ];
    }
}
```

The above code defines the `error` action using the [[yii\web\ErrorAction]] class which renders an error using a view named `error`.

Besides using [[yii\web\ErrorAction]], you may also define the `error` action using an action method like the following,

```
public function actionError()
{
    $exception = Yii::$app->errorHandler->exception;
    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}
```

You should now create a view file located at `views/site/error.php`. In this view file, you can access the following variables if the error action is defined as [[yii\web\ErrorAction]]:

- `name`: the name of the error;
- `message`: the error message;
- `exception`: the exception object through which you can retrieve more useful information, such as HTTP status code, error code, error call stack, etc.

> Info: If you are using the basic project template or the advanced project template, the error action and the error view are already defined for you.
>
> Note: If you need to redirect in an error handler, do it the following way:
>
> ```
> Yii::$app->getResponse()->redirect($url)->send();
> return;
> ```

Customizing Error Response Format

The error handler displays errors according to the format setting of the response. If the [[yii\web\Response::format|response format]] is `html`, it will use the error or exception view to display errors, as described in the last subsection. For other response formats, the error handler will assign the array representation of the exception to the [[yii\web\Response::data]] property which will then be converted to different formats accordingly. For example, if the response format is `json`, you may see the following response:

```
HTTP/1.1 404 Not Found
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

You may customize the error response format by responding to the `beforeSend` event of the `response` component in the application configuration:

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

The above code will reformat the error response like the following:

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "success": false,
    "data": {
        "name": "Not Found Exception",
        "message": "The requested resource was not found.",
        "code": 0,
        "status": 404
    }
}
```

Logging

Yii provides a powerful logging framework that is highly customizable and extensible. Using this framework, you can easily log various types of messages, filter them, and gather them at different targets, such as files, databases, emails.

Using the Yii logging framework involves the following steps:

- Record log messages at various places in your code;
- Configure log targets in the application configuration to filter and export log messages;
- Examine the filtered logged messages exported by different targets (e.g. the Yii debugger).

In this section, we will mainly describe the first two steps.

Log Messages

Recording log messages is as simple as calling one of the following logging methods:

- [[Yii::trace()]]: record a message to trace how a piece of code runs. This is mainly for development use.
- [[Yii::info()]]: record a message that conveys some useful information.
- [[Yii::warning()]]: record a warning message that indicates something unexpected has happened.
- [[Yii::error()]]: record a fatal error that should be investigated as soon as possible.

These logging methods record log messages at various *severity levels* and *categories*. They share the same function signature `function ($message, $category = 'application')`, where `$message` stands for the log message to be recorded, while `$category` is the category of the log message. The code in the following example records a trace message under the default category `application`:

```
Yii::trace('start calculating average revenue');
```

> Info: Log messages can be strings as well as complex data, such as arrays or objects. It is the responsibility of log targets to properly deal with log messages. By default, if a log message is not a string, it will be exported as a string by calling [[yii\helpers\VarDumper::export()]].

To better organize and filter log messages, it is recommended that you specify an appropriate category for each log message. You may choose a hierarchical naming scheme for categories, which will make it easier for log targets to filter messages based on their categories. A simple yet effective naming scheme is to use the PHP magic constant `__METHOD__` for the category names. This is also the approach used in the core Yii framework code. For example,

```
Yii::trace('start calculating average revenue', __METHOD__);
```

The `__METHOD__` constant evaluates as the name of the method (prefixed with the fully qualified class name) where the constant appears. For example, it is equal to the string `'app\controllers\RevenueController::calculate'` if the above line of code is called within this method.

> Info: The logging methods described above are actually shortcuts to the [[yii\log\Logger::log()|log()]] method of the [[yii\log\Logger|logger object]] which is a singleton accessible through the expression `Yii::getLogger()`. When enough messages are logged or when the application ends, the logger object will call a [[yii\log\Dispatcher|message dispatcher]] to send recorded log messages to the registered log targets.

Log Targets

A log target is an instance of the [[yii\log\Target]] class or its child class. It filters the log messages by their severity levels and categories and then exports them to some medium. For example, a [[yii\log\DbTarget|database target]] exports the filtered log messages to a database table, while an [[yii\log\EmailTarget|email target]] exports the log messages to specified email addresses.

You can register multiple log targets in an application by configuring them through the `log` application component in the application configuration, like the following:

```
return [
    // the "log" component must be loaded during bootstrapping time
    'bootstrap' => ['log'],

    'components' => [
        'log' => [
            'targets' => [
                [
                    'class' => 'yii\log\DbTarget',
                    'levels' => ['error', 'warning'],
                ],
                [
                    'class' => 'yii\log\EmailTarget',
                    'levels' => ['error'],
                    'categories' => ['yii\db\*'],
                    'message' => [
                       'from' => ['log@example.com'],
                       'to' => ['admin@example.com', 'developer@example.com'],
                       'subject' => 'Database errors at example.com',
                    ],
                ],
            ],
        ],
    ],
];
```

> Note: The `log` component must be loaded during bootstrapping time so that it can dispatch log messages to targets promptly. That is why it is listed in the `bootstrap` array as shown above.

In the above code, two log targets are registered in the [[yii\log\Dispatcher::targets]] property:

- the first target selects error and warning messages and saves them in a database table;
- the second target selects error messages under the categories whose names start with `yii\db\`, and sends them in an email to both `admin@example.com` and `developer@example.com`.

Yii comes with the following built-in log targets. Please refer to the API documentation about these classes to learn how to configure and use them.

- [[yii\log\DbTarget]]: stores log messages in a database table.
- [[yii\log\EmailTarget]]: sends log messages to pre-specified email addresses.
- [[yii\log\FileTarget]]: saves log messages in files.
- [[yii\log\SyslogTarget]]: saves log messages to syslog by calling the PHP function `syslog()`.

In the following, we will describe the features common to all log targets.

Message Filtering

For each log target, you can configure its [[yii\log\Target::levels|levels]] and [[yii\log\Target::categories|categories]] properties to specify which severity levels and categories of the messages the target should process.

The [[yii\log\Target::levels|levels]] property takes an array consisting of one or several of the following values:

- `error` : corresponding to messages logged by [[Yii::error()]].
- `warning` : corresponding to messages logged by [[Yii::warning()]].
- `info` : corresponding to messages logged by [[Yii::info()]].
- `trace` : corresponding to messages logged by [[Yii::trace()]].
- `profile` : corresponding to messages logged by [[Yii::beginProfile()]] and [[Yii::endProfile()]], which will be explained in more details in the Profiling subsection.

If you do not specify the [[yii\log\Target::levels|levels]] property, it means the target will process messages of *any* severity level.

The [[yii\log\Target::categories|categories]] property takes an array consisting of message category names or patterns. A target will only process messages whose category can be found or match one of the patterns in this array. A category pattern is a category name prefix with an asterisk `*` at its end. A category name matches a category pattern if it starts with the same prefix of the pattern. For example, `yii\db\Command::execute` and `yii\db\Command::query` are used as category names for the log messages recorded in the [[yii\db\Command]] class. They both match the pattern `yii\db\*` .

If you do not specify the [[yii\log\Target::categories|categories]] property, it means the target will process messages of *any* category.

Besides whitelisting the categories by the [[yii\log\Target::categories|categories]] property, you may also blacklist certain categories by the [[yii\log\Target::except|except]] property. If the category of a message is found or matches one of the patterns in this property, it will NOT be processed by the target.

The following target configuration specifies that the target should only process error and warning messages under the categories whose names match either `yii\db\*` or `yii\web\HttpException:*` , but not `yii\web\HttpException:404` .

```
[
    'class' => 'yii\log\FileTarget',
    'levels' => ['error', 'warning'],
    'categories' => [
        'yii\db\*',
        'yii\web\HttpException:*',
    ],
    'except' => [
        'yii\web\HttpException:404',
    ],
]
```

> Info: When an HTTP exception is caught by the error handler, an error message will be logged with the category name in the format of `yii\web\HttpException:ErrorCode`. For example, the [[yii\web\NotFoundHttpException]] will cause an error message of category `yii\web\HttpException:404`.

Message Formatting

Log targets export the filtered log messages in a certain format. For example, if you install a log target of the class [[yii\log\FileTarget]], you may find a log message similar to the following in the `runtime/log/app.log` file:

```
2014-10-04 18:10:15 [::1][][-][trace][yii\base\Module::getModule] Loading module: debu
g
```

By default, log messages will be formatted as follows by the [[yii\log\Target::formatMessage()]]:

```
Timestamp [IP address][User ID][Session ID][Severity Level][Category] Message Text
```

You may customize this format by configuring the [[yii\log\Target::prefix]] property which takes a PHP callable returning a customized message prefix. For example, the following code configures a log target to prefix each log message with the current user ID (IP address and Session ID are removed for privacy reasons).

```
[
    'class' => 'yii\log\FileTarget',
    'prefix' => function ($message) {
        $user = Yii::$app->has('user', true) ? Yii::$app->get('user') : null;
        $userID = $user ? $user->getId(false) : '-';
        return "[$userID]";
    }
]
```

Besides message prefixes, log targets also append some context information to each batch of log messages. By default, the values of these global PHP variables are included: `$_GET`, `$_POST`, `$_FILES`, `$_COOKIE`, `$_SESSION` and `$_SERVER`. You may adjust this behavior by configuring the [[yii\log\Target::logVars]] property with the names of the global variables that you want to include by the log target. For example, the following log target configuration specifies that only the value of the `$_SERVER` variable will be appended to the log messages.

```
[
    'class' => 'yii\log\FileTarget',
    'logVars' => ['_SERVER'],
]
```

You may configure `logVars` to be an empty array to totally disable the inclusion of context information. Or if you want to implement your own way of providing context information, you may override the [[yii\log\Target::getContextMessage()]] method.

Message Trace Level

During development, it is often desirable to see where each log message is coming from. This can be achieved by configuring the [[yii\log\Dispatcher::traceLevel|traceLevel]] property of the `log` component like the following:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [...],
        ],
    ],
];
```

The above application configuration sets [[yii\log\Dispatcher::traceLevel|traceLevel]] to be 3 if `YII_DEBUG` is on and 0 if `YII_DEBUG` is off. This means, if `YII_DEBUG` is on, each log message will be appended with at most 3 levels of the call stack at which the log message is

recorded; and if `YII_DEBUG` is off, no call stack information will be included.

> Info: Getting call stack information is not trivial. Therefore, you should only use this feature during development or when debugging an application.

Message Flushing and Exporting

As aforementioned, log messages are maintained in an array by the [[yii\log\Logger|logger object]]. To limit the memory consumption by this array, the logger will flush the recorded messages to the log targets each time the array accumulates a certain number of log messages. You can customize this number by configuring the [[yii\log\Dispatcher::flushInterval|flushInterval]] property of the `log` component:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 100,   // default is 1000
            'targets' => [...],
        ],
    ],
];
```

> Info: Message flushing also occurs when the application ends, which ensures log targets can receive complete log messages.

When the [[yii\log\Logger|logger object]] flushes log messages to log targets, they do not get exported immediately. Instead, the message exporting only occurs when a log target accumulates certain number of the filtered messages. You can customize this number by configuring the [[yii\log\Target::exportInterval|exportInterval]] property of individual log targets, like the following,

```
[
    'class' => 'yii\log\FileTarget',
    'exportInterval' => 100,  // default is 1000
]
```

Because of the flushing and exporting level setting, by default when you call `Yii::trace()` or any other logging method, you will NOT see the log message immediately in the log targets. This could be a problem for some long-running console applications. To make each log message appear immediately in the log targets, you should set both [[yii\log\Dispatcher::flushInterval|flushInterval]] and [[yii\log\Target::exportInterval|exportInterval]] to be 1, as shown below:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'flushInterval' => 1,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                    'exportInterval' => 1,
                ],
            ],
        ],
    ],
];
```

> Note: Frequent message flushing and exporting will degrade the performance of your application.

Toggling Log Targets

You can enable or disable a log target by configuring its [[yii\log\Target::enabled|enabled]] property. You may do so via the log target configuration or by the following PHP statement in your code:

```
Yii::$app->log->targets['file']->enabled = false;
```

The above code requires you to name a target as `file`, as shown below by using string keys in the `targets` array:

```
return [
    'bootstrap' => ['log'],
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                ],
                'db' => [
                    'class' => 'yii\log\DbTarget',
                ],
            ],
        ],
    ],
];
```

Creating New Targets

Creating a new log target class is very simple. You mainly need to implement the [[yii\log\Target::export()]] method sending the content of the [[yii\log\Target::messages]] array to a designated medium. You may call the [[yii\log\Target::formatMessage()]] method to format each message. For more details, you may refer to any of the log target classes included in the Yii release.

Performance Profiling

Performance profiling is a special type of message logging that is used to measure the time taken by certain code blocks and find out what are the performance bottlenecks. For example, the [[yii\db\Command]] class uses performance profiling to find out the time taken by each DB query.

To use performance profiling, first identify the code blocks that need to be profiled. Then enclose each code block like the following:

```
\Yii::beginProfile('myBenchmark');

...code block being profiled...

\Yii::endProfile('myBenchmark');
```

where `myBenchmark` stands for a unique token identifying a code block. Later when you examine the profiling result, you will use this token to locate the time spent by the corresponding code block.

It is important to make sure that the pairs of `beginProfile` and `endProfile` are properly nested. For example,

```
\Yii::beginProfile('block1');

    // some code to be profiled

    \Yii::beginProfile('block2');
        // some other code to be profiled
    \Yii::endProfile('block2');

\Yii::endProfile('block1');
```

If you miss `\Yii::endProfile('block1')` or switch the order of `\Yii::endProfile('block1')` and `\Yii::endProfile('block2')`, the performance profiling will not work.

For each code block being profiled, a log message with the severity level `profile` is recorded. You can configure a log target to collect such messages and export them. The Yii debugger has a built-in performance profiling panel showing the profiling results.

Components

Components are the main building blocks of Yii applications. Components are instances of [[yii\base\Component]], or an extended class. The three main features that components provide to other classes are:

- Properties
- Events
- Behaviors

Separately and combined, these features make Yii classes much more customizable and easier to use. For example, the included [[yii\jui\DatePicker|date picker widget]], a user interface component, can be used in a view to generate an interactive date picker:

```
use yii\jui\DatePicker;

echo DatePicker::widget([
    'language' => 'ru',
    'name'  => 'country',
    'clientOptions' => [
        'dateFormat' => 'yy-mm-dd',
    ],
]);
```

The widget's properties are easily writable because the class extends [[yii\base\Component]].

While components are very powerful, they are a bit heavier than normal objects, due to the fact that it takes extra memory and CPU time to support event and behavior functionality in particular. If your components do not need these two features, you may consider extending your component class from [[yii\base\Object]] instead of [[yii\base\Component]]. Doing so will make your components as efficient as normal PHP objects, but with added support for properties.

When extending your class from [[yii\base\Component]] or [[yii\base\Object]], it is recommended that you follow these conventions:

- If you override the constructor, specify a `$config` parameter as the constructor's *last* parameter, and then pass this parameter to the parent constructor.
- Always call the parent constructor *at the end* of your overriding constructor.
- If you override the [[yii\base\Object::init()]] method, make sure you call the parent implementation of `init()` *at the beginning* of your `init()` method.

For example:

```
<?php

namespace yii\components\MyClass;

use yii\base\Object;

class MyClass extends Object
{
    public $prop1;
    public $prop2;

    public function __construct($param1, $param2, $config = [])
    {
        // ... initialization before configuration is applied

        parent::__construct($config);
    }

    public function init()
    {
        parent::init();

        // ... initialization after configuration is applied
    }
}
```

Following these guidelines will make your components configurable when they are created. For example:

```
$component = new MyClass(1, 2, ['prop1' => 3, 'prop2' => 4]);
// alternatively
$component = \Yii::createObject([
    'class' => MyClass::className(),
    'prop1' => 3,
    'prop2' => 4,
], [1, 2]);
```

> Info: While the approach of calling [[Yii::createObject()]] looks more complicated, it is more powerful because it is implemented on top of a dependency injection container.

The [[yii\base\Object]] class enforces the following object lifecycle:

1. Pre-initialization within the constructor. You can set default property values here.
2. Object configuration via `$config`. The configuration may overwrite the default values set within the constructor.
3. Post-initialization within [[yii\base\Object::init()|init()]]. You may override this method to

perform sanity checks and normalization of the properties.
4. Object method calls.

The first three steps all happen within the object's constructor. This means that once you get a class instance (i.e., an object), that object has already been initialized to a proper, reliable state.

Properties

In PHP, class member variables are also called *properties*. These variables are part of the class definition, and are used to represent the state of a class instance (i.e., to differentiate one instance of the class from another). In practice, you may often want to handle the reading or writing of properties in special ways. For example, you may want to always trim a string when it is being assigned to a `label` property. You *could* use the following code to achieve this task:

```
$object->label = trim($label);
```

The drawback of the above code is that you would have to call `trim()` everywhere in your code where you might set the `label` property. If, in the future, the `label` property gets a new requirement, such as the first letter must be capitalized, you would again have to modify every bit of code that assigns a value to `label`. The repetition of code leads to bugs, and is a practice you want to avoid as much as possible.

To solve this problem, Yii introduces a base class called [[yii\base\Object]] that supports defining properties based on *getter* and *setter* class methods. If a class needs that functionality, it should extend from [[yii\base\Object]], or from a child class.

> Info: Nearly every core class in the Yii framework extends from [[yii\base\Object]] or a child class. This means, that whenever you see a getter or setter in a core class, you can use it like a property.

A getter method is a method whose name starts with the word `get`; a setter method starts with `set`. The name after the `get` or `set` prefix defines the name of a property. For example, a getter `getLabel()` and/or a setter `setLabel()` defines a property named `label`, as shown in the following code:

```
namespace app\components;

use yii\base\Object;

class Foo extends Object
{
    private $_label;

    public function getLabel()
    {
        return $this->_label;
    }

    public function setLabel($value)
    {
        $this->_label = trim($value);
    }
}
```

To be clear, the getter and setter methods create the property `label`, which in this case internally refers to a private property named `_label`.

Properties defined by getters and setters can be used like class member variables. The main difference is that when such property is being read, the corresponding getter method will be called; when the property is being assigned a value, the corresponding setter method will be called. For example:

```
// equivalent to $label = $object->getLabel();
$label = $object->label;

// equivalent to $object->setLabel('abc');
$object->label = 'abc';
```

A property defined by a getter without a setter is *read only*. Trying to assign a value to such a property will cause an [[yii\base\InvalidCallException|InvalidCallException]]. Similarly, a property defined by a setter without a getter is *write only*, and trying to read such a property will also cause an exception. It is not common to have write-only properties.

There are several special rules for, and limitations on, the properties defined via getters and setters:

- The names of such properties are *case-insensitive*. For example, `$object->label` and `$object->Label` are the same. This is because method names in PHP are case-insensitive.
- If the name of such a property is the same as a class member variable, the latter will take precedence. For example, if the above `Foo` class has a member variable `label`,

then the assignment `$object->label = 'abc'` will affect the *member variable* `label`; that line would not call the `setLabel()` setter method.
- These properties do not support visibility. It makes no difference to the defining getter or setter method if the property is public, protected or private.
- The properties can only be defined by *non-static* getters and/or setters. Static methods will not be treated in the same manner.
- A normal call to `property_exists()` does not work to determine magic properties. You should call [[yii\base\Object::canGetProperty()|canGetProperty()]] or [[yii\base\Object::canSetProperty()|canSetProperty()]] respectively.

Returning back to the problem described at the beginning of this guide, instead of calling `trim()` everywhere a `label` value is assigned, `trim()` now only needs to be invoked within the setter `setLabel()`. And if a new requirement makes it necessary that the label be initially capitalized, the `setLabel()` method can quickly be modified without touching any other code. The one change will universally affect every assignment to `label`.

Events

Events allow you to inject custom code into existing code at certain execution points. You can attach custom code to an event so that when the event is triggered, the code gets executed automatically. For example, a mailer object may trigger a `messageSent` event when it successfully sends a message. If you want to keep track of the messages that are successfully sent, you could then simply attach the tracking code to the `messageSent` event.

Yii introduces a base class called [[yii\base\Component]] to support events. If a class needs to trigger events, it should extend from [[yii\base\Component]], or from a child class.

Event Handlers

An event handler is a PHP callback that gets executed when the event it is attached to is triggered. You can use any of the following callbacks:

- a global PHP function specified as a string (without parentheses), e.g., `'trim'`;
- an object method specified as an array of an object and a method name as a string (without parentheses), e.g., `[$object, 'methodName']`;
- a static class method specified as an array of a class name and a method name as a string (without parentheses), e.g., `['ClassName', 'methodName']`;
- an anonymous function, e.g., `function ($event) { ... }`.

The signature of an event handler is:

```
function ($event) {
    // $event is an object of yii\base\Event or a child class
}
```

Through the `$event` parameter, an event handler may get the following information about the event that occurred:

- [[yii\base\Event::name|event name]];
- [[yii\base\Event::sender|event sender]]: the object whose `trigger()` method was called;
- [[yii\base\Event::data|custom data]]: the data that is provided when attaching the event handler (to be explained next).

Attaching Event Handlers

You can attach a handler to an event by calling the [[yii\base\Component::on()]] method. For example:

```
$foo = new Foo;

// this handler is a global function
$foo->on(Foo::EVENT_HELLO, 'function_name');

// this handler is an object method
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);

// this handler is a static class method
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// this handler is an anonymous function
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // event handling logic
});
```

You may also attach event handlers through configurations. For more details, please refer to the Configurations section.

When attaching an event handler, you may provide additional data as the third parameter to [[yii\base\Component::on()]]. The data will be made available to the handler when the event is triggered and the handler is called. For example:

```
// The following code will display "abc" when the event is triggered
// because $event->data contains the data passed as the 3rd argument to "on"
$foo->on(Foo::EVENT_HELLO, 'function_name', 'abc');

function function_name($event) {
    echo $event->data;
}
```

Event Handler Order

You may attach one or more handlers to a single event. When an event is triggered, the attached handlers will be called in the order that they were attached to the event. If a handler needs to stop the invocation of the handlers that follow it, it may set the [[yii\base\Event::handled]] property of the `$event` parameter to be `true`:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    $event->handled = true;
});
```

By default, a newly attached handler is appended to the existing handler queue for the event. As a result, the handler will be called in the last place when the event is triggered. To insert the new handler at the start of the handler queue so that the handler gets called first, you may call [[yii\base\Component::on()]], passing `false` for the fourth parameter `$append`:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // ...
}, $data, false);
```

Triggering Events

Events are triggered by calling the [[yii\base\Component::trigger()]] method. The method requires an *event name*, and optionally an event object that describes the parameters to be passed to the event handlers. For example:

```
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class Foo extends Component
{
    const EVENT_HELLO = 'hello';

    public function bar()
    {
        $this->trigger(self::EVENT_HELLO);
    }
}
```

With the above code, any calls to `bar()` will trigger an event named `hello`.

> Tip: It is recommended to use class constants to represent event names. In the above example, the constant `EVENT_HELLO` represents the `hello` event. This approach has three benefits. First, it prevents typos. Second, it can make events recognizable for IDE auto-completion support. Third, you can tell what events are supported in a class by simply checking its constant declarations.

Sometimes when triggering an event you may want to pass along additional information to the event handlers. For example, a mailer may want to pass the message information to the handlers of the `messageSent` event so that the handlers can know the particulars of the sent

messages. To do so, you can provide an event object as the second parameter to the [[yii\base\Component::trigger()]] method. The event object must be an instance of the [[yii\base\Event]] class or a child class. For example:

```
namespace app\components;

use yii\base\Component;
use yii\base\Event;

class MessageEvent extends Event
{
    public $message;
}

class Mailer extends Component
{
    const EVENT_MESSAGE_SENT = 'messageSent';

    public function send($message)
    {
        // ...sending $message...

        $event = new MessageEvent;
        $event->message = $message;
        $this->trigger(self::EVENT_MESSAGE_SENT, $event);
    }
}
```

When the [[yii\base\Component::trigger()]] method is called, it will call all handlers attached to the named event.

Detaching Event Handlers

To detach a handler from an event, call the [[yii\base\Component::off()]] method. For example:

```
// the handler is a global function
$foo->off(Foo::EVENT_HELLO, 'function_name');

// the handler is an object method
$foo->off(Foo::EVENT_HELLO, [$object, 'methodName']);

// the handler is a static class method
$foo->off(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);

// the handler is an anonymous function
$foo->off(Foo::EVENT_HELLO, $anonymousFunction);
```

Note that in general you should not try to detach an anonymous function unless you store it somewhere when it is attached to the event. In the above example, it is assumed that the anonymous function is stored as a variable `$anonymousFunction`.

To detach *all* handlers from an event, simply call [[yii\base\Component::off()]] without the second parameter:

```
$foo->off(Foo::EVENT_HELLO);
```

Class-Level Event Handlers

The above subsections described how to attach a handler to an event on an *instance level*. Sometimes, you may want to respond to an event triggered by *every* instance of a class instead of only by a specific instance. Instead of attaching an event handler to every instance, you may attach the handler on the *class level* by calling the static method [[yii\base\Event::on()]].

For example, an Active Record object will trigger an [[yii\db\BaseActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]] event whenever it inserts a new record into the database. In order to track insertions done by *every* Active Record object, you may use the following code:

```
use Yii;
use yii\base\Event;
use yii\db\ActiveRecord;

Event::on(ActiveRecord::className(), ActiveRecord::EVENT_AFTER_INSERT, function ($even
t) {
    Yii::trace(get_class($event->sender) . ' is inserted');
});
```

The event handler will be invoked whenever an instance of [[yii\db\ActiveRecord|ActiveRecord]], or one of its child classes, triggers the [[yii\db\BaseActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]] event. In the handler, you can get the object that triggered the event through `$event->sender`.

When an object triggers an event, it will first call instance-level handlers, followed by the class-level handlers.

You may trigger a *class-level* event by calling the static method [[yii\base\Event::trigger()]]. A class-level event is not associated with a particular object. As a result, it will cause the invocation of class-level event handlers only. For example:

```
use yii\base\Event;

Event::on(Foo::className(), Foo::EVENT_HELLO, function ($event) {
    var_dump($event->sender);  // displays "null"
});

Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

Note that, in this case, `$event->sender` is `null` instead of an object instance.

> Note: Because a class-level handler will respond to an event triggered by any instance of that class, or any child classes, you should use it carefully, especially if the class is a low-level base class, such as [[yii\base\Object]].

To detach a class-level event handler, call [[yii\base\Event::off()]]. For example:

```
// detach $handler
Event::off(Foo::className(), Foo::EVENT_HELLO, $handler);

// detach all handlers of Foo::EVENT_HELLO
Event::off(Foo::className(), Foo::EVENT_HELLO);
```

Events using interfaces

There is even more abstract way to deal with events. You can create a separated interface for the special event and implement it in classes, where you need it.

For example, we can create the following interface:

```
namespace app\interfaces;

interface DanceEventInterface
{
    const EVENT_DANCE = 'dance';
}
```

And two classes, that implement it:

```
class Dog extends Component implements DanceEventInterface
{
    public function meetBuddy()
    {
        echo "Woof!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}

class Developer extends Component implements DanceEventInterface
{
    public function testsPassed()
    {
        echo "Yay!";
        $this->trigger(DanceEventInterface::EVENT_DANCE);
    }
}
```

To handle the `EVENT_DANCE`, triggered by any of these classes, call [[yii\base\Event::on()|Event::on()]] and pass the interface class name as the first argument:

```
Event::on('app\interfaces\DanceEventInterface', DanceEventInterface::EVENT_DANCE, func
tion ($event) {
    Yii::trace(get_class($event->sender) . ' just danced'); // Will log that Dog or De
veloper danced
});
```

You can trigger the event of those classes:

```
// trigger event for Dog class
Event::trigger(Dog::className(), DanceEventInterface::EVENT_DANCE);

// trigger event for Developer class
Event::trigger(Developer::className(), DanceEventInterface::EVENT_DANCE);
```

But please notice, that you can not trigger all the classes, that implement the interface:

```
// DOES NOT WORK. Classes that implement this interface will NOT be triggered.
Event::trigger('app\interfaces\DanceEventInterface', DanceEventInterface::EVENT_DANCE)
;
```

To detach event handler, call [[yii\base\Event::off()|Event::off()]]. For example:

```
// detaches $handler
Event::off('app\interfaces\DanceEventInterface', DanceEventInterface::EVENT_DANCE, $ha
ndler);

// detaches all handlers of DanceEventInterface::EVENT_DANCE
Event::off('app\interfaces\DanceEventInterface', DanceEventInterface::EVENT_DANCE);
```

Global Events

Yii supports a so-called *global event*, which is actually a trick based on the event mechanism described above. The global event requires a globally accessible Singleton, such as the application instance itself.

To create the global event, an event sender calls the Singleton's `trigger()` method to trigger the event, instead of calling the sender's own `trigger()` method. Similarly, the event handlers are attached to the event on the Singleton. For example:

```
use Yii;
use yii\base\Event;
use app\components\Foo;

Yii::$app->on('bar', function ($event) {
    echo get_class($event->sender);  // displays "app\components\Foo"
});

Yii::$app->trigger('bar', new Event(['sender' => new Foo]));
```

A benefit of using global events is that you do not need an object when attaching a handler to the event which will be triggered by the object. Instead, the handler attachment and the event triggering are both done through the Singleton (e.g. the application instance).

However, because the namespace of the global events is shared by all parties, you should name the global events wisely, such as introducing some sort of namespace (e.g. "frontend.mail.sent", "backend.mail.sent").

Behaviors

Behaviors are instances of [[yii\base\Behavior]], or of a child class. Behaviors, also known as mixins, allow you to enhance the functionality of an existing [[yii\base\Component|component]] class without needing to change the class's inheritance. Attaching a behavior to a component "injects" the behavior's methods and properties into the component, making those methods and properties accessible as if they were defined in the component class itself. Moreover, a behavior can respond to the events triggered by the component, which allows behaviors to also customize the normal code execution of the component.

Defining Behaviors

To define a behavior, create a class that extends [[yii\base\Behavior]], or extends a child class. For example:

```
namespace app\components;

use yii\base\Behavior;

class MyBehavior extends Behavior
{
    public $prop1;

    private $_prop2;

    public function getProp2()
    {
        return $this->_prop2;
    }

    public function setProp2($value)
    {
        $this->_prop2 = $value;
    }

    public function foo()
    {
        // ...
    }
}
```

The above code defines the behavior class `app\components\MyBehavior`, with two properties--`prop1` and `prop2`--and one method `foo()`. Note that property `prop2` is defined via the getter `getProp2()` and the setter `setProp2()`. This is the case because [[yii\base\Behavior]] extends [[yii\base\Object]] and therefore supports defining properties via getters and setters.

Because this class is a behavior, when it is attached to a component, that component will then also have the `prop1` and `prop2` properties and the `foo()` method.

> Tip: Within a behavior, you can access the component that the behavior is attached to through the [[yii\base\Behavior::owner]] property.
>
> Note: In case [[yii\base\Behavior::**get()]] and/or [[yii\base\Behavior::**set()]] method of behavior is overridden you need to override [[yii\base\Behavior::canGetProperty()]] and/or [[yii\base\Behavior::canSetProperty()]] as well.

Handling Component Events

If a behavior needs to respond to the events triggered by the component it is attached to, it should override the [[yii\base\Behavior::events()]] method. For example:

```
namespace app\components;

use yii\db\ActiveRecord;
use yii\base\Behavior;

class MyBehavior extends Behavior
{
    // ...

    public function events()
    {
        return [
            ActiveRecord::EVENT_BEFORE_VALIDATE => 'beforeValidate',
        ];
    }

    public function beforeValidate($event)
    {
        // ...
    }
}
```

The [[yii\base\Behavior::events()|events()]] method should return a list of events and their corresponding handlers. The above example declares that the [[yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE|EVENT_BEFORE_VALIDATE]] event

exists and defines its handler, `beforeValidate()`. When specifying an event handler, you may use one of the following formats:

- a string that refers to the name of a method of the behavior class, like the example above
- an array of an object or class name, and a method name as a string (without parentheses), e.g., `[$object, 'methodName']`;
- an anonymous function

The signature of an event handler should be as follows, where `$event` refers to the event parameter. Please refer to the Events section for more details about events.

```
function ($event) {
}
```

Attaching Behaviors

You can attach a behavior to a [[yii\base\Component|component]] either statically or dynamically. The former is more common in practice.

To attach a behavior statically, override the [[yii\base\Component::behaviors()|behaviors()]] method of the component class to which the behavior is being attached. The [[yii\base\Component::behaviors()|behaviors()]] method should return a list of behavior configurations. Each behavior configuration can be either a behavior class name or a configuration array:

```php
namespace app\models;

use yii\db\ActiveRecord;
use app\components\MyBehavior;

class User extends ActiveRecord
{
    public function behaviors()
    {
        return [
            // anonymous behavior, behavior class name only
            MyBehavior::className(),

            // named behavior, behavior class name only
            'myBehavior2' => MyBehavior::className(),

            // anonymous behavior, configuration array
            [
                'class' => MyBehavior::className(),
                'prop1' => 'value1',
                'prop2' => 'value2',
            ],

            // named behavior, configuration array
            'myBehavior4' => [
                'class' => MyBehavior::className(),
                'prop1' => 'value1',
                'prop2' => 'value2',
            ]
        ];
    }
}
```

You may associate a name with a behavior by specifying the array key corresponding to the behavior configuration. In this case, the behavior is called a *named behavior*. In the above example, there are two named behaviors: `myBehavior2` and `myBehavior4`. If a behavior is not associated with a name, it is called an *anonymous behavior*.

To attach a behavior dynamically, call the [[yii\base\Component::attachBehavior()]] method of the component to which the behavior is being attached:

```
use app\components\MyBehavior;

// attach a behavior object
$component->attachBehavior('myBehavior1', new MyBehavior);

// attach a behavior class
$component->attachBehavior('myBehavior2', MyBehavior::className());

// attach a configuration array
$component->attachBehavior('myBehavior3', [
    'class' => MyBehavior::className(),
    'prop1' => 'value1',
    'prop2' => 'value2',
]);
```

You may attach multiple behaviors at once using the [[yii\base\Component::attachBehaviors()]] method:

```
$component->attachBehaviors([
    'myBehavior1' => new MyBehavior,  // a named behavior
    MyBehavior::className(),          // an anonymous behavior
]);
```

You may also attach behaviors through configurations like the following:

```
[
    'as myBehavior2' => MyBehavior::className(),

    'as myBehavior3' => [
        'class' => MyBehavior::className(),
        'prop1' => 'value1',
        'prop2' => 'value2',
    ],
]
```

For more details, please refer to the Configurations section.

Using Behaviors

To use a behavior, first attach it to a [[yii\base\Component|component]] per the instructions above. Once a behavior is attached to a component, its usage is straightforward.

You can access a *public* member variable or a property defined by a getter and/or a setter of the behavior through the component it is attached to:

```
// "prop1" is a property defined in the behavior class
echo $component->prop1;
$component->prop1 = $value;
```

You can also call a *public* method of the behavior similarly:

```
// foo() is a public method defined in the behavior class
$component->foo();
```

As you can see, although `$component` does not define `prop1` and `foo()`, they can be used as if they are part of the component definition due to the attached behavior.

If two behaviors define the same property or method and they are both attached to the same component, the behavior that is attached to the component *first* will take precedence when the property or method is accessed.

A behavior may be associated with a name when it is attached to a component. If this is the case, you may access the behavior object using the name:

```
$behavior = $component->getBehavior('myBehavior');
```

You may also get all behaviors attached to a component:

```
$behaviors = $component->getBehaviors();
```

Detaching Behaviors

To detach a behavior, call [[yii\base\Component::detachBehavior()]] with the name associated with the behavior:

```
$component->detachBehavior('myBehavior1');
```

You may also detach *all* behaviors:

```
$component->detachBehaviors();
```

Using `TimestampBehavior`

To wrap up, let's take a look at [[yii\behaviors\TimestampBehavior]]. This behavior supports automatically updating the timestamp attributes of an [[yii\db\ActiveRecord|Active Record]] model anytime the model is saved via `insert()`, `update()` or `save()` method.

First, attach this behavior to the [[yii\db\ActiveRecord|Active Record]] class that you plan to use:

```
namespace app\models\User;

use yii\db\ActiveRecord;
use yii\behaviors\TimestampBehavior;

class User extends ActiveRecord
{
    // ...

    public function behaviors()
    {
        return [
            [
                'class' => TimestampBehavior::className(),
                'attributes' => [
                    ActiveRecord::EVENT_BEFORE_INSERT => ['created_at', 'updated_at'],
                    ActiveRecord::EVENT_BEFORE_UPDATE => ['updated_at'],
                ],
                // if you're using datetime instead of UNIX timestamp:
                // 'value' => new Expression('NOW()'),
            ],
        ];
    }
}
```

The behavior configuration above specifies that when the record is being:

- inserted, the behavior should assign the current UNIX timestamp to the `created_at` and `updated_at` attributes
- updated, the behavior should assign the current UNIX timestamp to the `updated_at` attribute

> Note: For the above implementation to work with MySQL database, please declare the columns(`created_at`, `updated_at`) as int(11) for being UNIX timestamp.

With that code in place, if you have a `User` object and try to save it, you will find its `created_at` and `updated_at` are automatically filled with the current UNIX timestamp:

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at;  // shows the current timestamp
```

The [[yii\behaviors\TimestampBehavior|TimestampBehavior]] also offers a useful method [[yii\behaviors\TimestampBehavior::touch()|touch()]], which will assign the current timestamp to a specified attribute and save it to the database:

```
$user->touch('login_time');
```

Other behaviors

There are several built-in and external behaviors available:

- [[yii\behaviors\BlameableBehavior]] - automatically fills the specified attributes with the current user ID.
- [[yii\behaviors\SluggableBehavior]] - automatically fills the specified attribute with a value that can be used as a slug in a URL.
- [[yii\behaviors\AttributeBehavior]] - automatically assigns a specified value to one or multiple attributes of an ActiveRecord object when certain events happen.
- yii2tech\ar\softdelete\SoftDeleteBehavior - provides methods to soft-delete and soft-restore ActiveRecord i.e. set flag or status which marks record as deleted.
- yii2tech\ar\position\PositionBehavior - allows managing records order in an integer field by providing reordering methods.

Comparing Behaviors with Traits

While behaviors are similar to traits in that they both "inject" their properties and methods to the primary class, they differ in many aspects. As explained below, they both have pros and cons. They are more like complements to each other rather than alternatives.

Reasons to Use Behaviors

Behavior classes, like normal classes, support inheritance. Traits, on the other hand, can be considered as language-supported copy and paste. They do not support inheritance.

Behaviors can be attached and detached to a component dynamically without requiring modification of the component class. To use a trait, you must modify the code of the class using it.

Behaviors are configurable while traits are not.

Behaviors can customize the code execution of a component by responding to its events.

When there can be name conflicts among different behaviors attached to the same component, the conflicts are automatically resolved by prioritizing the behavior attached to the component first. Name conflicts caused by different traits requires manual resolution by renaming the affected properties or methods.

Reasons to Use Traits

Traits are much more efficient than behaviors as behaviors are objects that take both time and memory.

IDEs are more friendly to traits as they are a native language construct.

Configurations

Configurations are widely used in Yii when creating new objects or initializing existing objects. Configurations usually include the class name of the object being created, and a list of initial values that should be assigned to the object's properties. Configurations may also include a list of handlers that should be attached to the object's events and/or a list of behaviors that should also be attached to the object.

In the following, a configuration is used to create and initialize a database connection:

```
$config = [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
];

$db = Yii::createObject($config);
```

The [[Yii::createObject()]] method takes a configuration array as its argument, and creates an object by instantiating the class named in the configuration. When the object is instantiated, the rest of the configuration will be used to initialize the object's properties, event handlers, and behaviors.

If you already have an object, you may use [[Yii::configure()]] to initialize the object's properties with a configuration array:

```
Yii::configure($object, $config);
```

Note that, in this case, the configuration array should not contain a `class` element.

Configuration Format

The format of a configuration can be formally described as:

```
[
    'class' => 'ClassName',
    'propertyName' => 'propertyValue',
    'on eventName' => $eventHandler,
    'as behaviorName' => $behaviorConfig,
]
```

where

- The `class` element specifies a fully qualified class name for the object being created.
- The `propertyName` elements specify the initial values for the named property. The keys are the property names, and the values are the corresponding initial values. Only public member variables and properties defined by getters/setters can be configured.
- The `on eventName` elements specify what handlers should be attached to the object's events. Notice that the array keys are formed by prefixing event names with `on`. Please refer to the Events section for supported event handler formats.
- The `as behaviorName` elements specify what behaviors should be attached to the object. Notice that the array keys are formed by prefixing behavior names with `as`; the value, `$behaviorConfig`, represents the configuration for creating a behavior, like a normal configuration described here.

Below is an example showing a configuration with initial property values, event handlers, and behaviors:

```
[
    'class' => 'app\components\SearchEngine',
    'apiKey' => 'xxxxxxxx',
    'on search' => function ($event) {
        Yii::info("Keyword searched: " . $event->keyword);
    },
    'as indexer' => [
        'class' => 'app\components\IndexerBehavior',
        // ... property init values ...
    ],
]
```

Using Configurations

Configurations are used in many places in Yii. At the beginning of this section, we have shown how to create an object according to a configuration by using [[Yii::createObject()]]. In this subsection, we will describe application configurations and widget configurations - two major usages of configurations.

Application Configurations

The configuration for an application is probably one of the most complex arrays in Yii. This is because the [[yii\web\Application|application]] class has a lot of configurable properties and events. More importantly, its [[yii\web\Application::components|components]] property can receive an array of configurations for creating components that are registered through the application. The following is an abstract from the application configuration file for the Basic Project Template.

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php'),
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
        'log' => [
            'class' => 'yii\log\Dispatcher',
            'traceLevel' => YII_DEBUG ? 3 : 0,
            'targets' => [
                [
                    'class' => 'yii\log\FileTarget',
                ],
            ],
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=stay2',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
];
```

The configuration does not have a `class` key. This is because it is used as follows in an entry script, where the class name is already given,

```
(new yii\web\Application($config))->run();
```

More details about configuring the `components` property of an application can be found in the Applications section and the Service Locator section.

Since version 2.0.11, the application configuration supports Dependency Injection Container configuration using `container` property. For example:

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php'),
    'container' => [
        'definitions' => [
            'yii\widgets\LinkPager' => ['maxButtonCount' => 5]
        ],
        'singletons' => [
            // Dependency Injection Container singletons configuration
        ]
    ]
];
```

To know more about the possible values of `definitions` and `singletons` configuration arrays and real-life examples, please read Advanced Practical Usage subsection of the Dependency Injection Container article.

Widget Configurations

When using widgets, you often need to use configurations to customize the widget properties. Both of the [[yii\base\Widget::widget()]] and [[yii\base\Widget::begin()]] methods can be used to create a widget. They take a configuration array, like the following,

```
use yii\widgets\Menu;

echo Menu::widget([
    'activateItems' => false,
    'items' => [
        ['label' => 'Home', 'url' => ['site/index']],
        ['label' => 'Products', 'url' => ['product/index']],
        ['label' => 'Login', 'url' => ['site/login'], 'visible' => Yii::$app->user->is
Guest],
    ],
]);
```

The above code creates a `Menu` widget and initializes its `activateItems` property to be `false`. The `items` property is also configured with menu items to be displayed.

Note that because the class name is already given, the configuration array should NOT have the `class` key.

Configuration Files

When a configuration is very complex, a common practice is to store it in one or multiple PHP files, known as *configuration files*. A configuration file returns a PHP array representing the configuration. For example, you may keep an application configuration in a file named `web.php`, like the following,

```
return [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php'),
    'components' => require(__DIR__ . '/components.php'),
];
```

Because the `components` configuration is complex too, you store it in a separate file called `components.php` and "require" this file in `web.php` as shown above. The content of `components.php` is as follows,

```
return [
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
    'mailer' => [
        'class' => 'yii\swiftmailer\Mailer',
    ],
    'log' => [
        'class' => 'yii\log\Dispatcher',
        'traceLevel' => YII_DEBUG ? 3 : 0,
        'targets' => [
            [
                'class' => 'yii\log\FileTarget',
            ],
        ],
    ],
    'db' => [
        'class' => 'yii\db\Connection',
        'dsn' => 'mysql:host=localhost;dbname=stay2',
        'username' => 'root',
        'password' => '',
        'charset' => 'utf8',
    ],
];
```

To get a configuration stored in a configuration file, simply "require" it, like the following:

```
$config = require('path/to/web.php');
(new yii\web\Application($config))->run();
```

Default Configurations

The [[Yii::createObject()]] method is implemented based on a dependency injection container. It allows you to specify a set of the so-called *default configurations* which will be applied to ALL instances of the specified classes when they are being created using [[Yii::createObject()]]. The default configurations can be specified by calling `Yii::$container->set()` in the bootstrapping code.

For example, if you want to customize [[yii\widgets\LinkPager]] so that ALL link pagers will show at most 5 page buttons (the default value is 10), you may use the following code to achieve this goal:

```
\Yii::$container->set('yii\widgets\LinkPager', [
    'maxButtonCount' => 5,
]);
```

Without using default configurations, you would have to configure `maxButtonCount` in every place where you use link pagers.

Environment Constants

Configurations often vary according to the environment in which an application runs. For example, in development environment, you may want to use a database named `mydb_dev`, while on production server you may want to use the `mydb_prod` database. To facilitate switching environments, Yii provides a constant named `YII_ENV` that you may define in the entry script of your application. For example,

```
defined('YII_ENV') or define('YII_ENV', 'dev');
```

You may define `YII_ENV` as one of the following values:

- `prod`: production environment. The constant `YII_ENV_PROD` will evaluate as `true`. This is the default value of `YII_ENV` if you do not define it.
- `dev`: development environment. The constant `YII_ENV_DEV` will evaluate as `true`.
- `test`: testing environment. The constant `YII_ENV_TEST` will evaluate as `true`.

With these environment constants, you may specify your configurations conditionally based on the current environment. For example, your application configuration may contain the following code to enable the debug toolbar and debugger in development environment.

```
$config = [...];

if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';
}

return $config;
```

Aliases

Aliases are used to represent file paths or URLs so that you don't have to hard-code absolute paths or URLs in your project. An alias must start with the `@` character to be differentiated from normal file paths and URLs. Alias defined without leading `@` will be prefixed with `@` character.

Yii has many pre-defined aliases already available. For example, the alias `@yii` represents the installation path of the Yii framework; `@web` represents the base URL for the currently running Web application.

Defining Aliases

You can define an alias for a file path or URL by calling [[Yii::setAlias()]]:

```
// an alias of a file path
Yii::setAlias('@foo', '/path/to/foo');

// an alias of a URL
Yii::setAlias('@bar', 'http://www.example.com');
```

> Note: The file path or URL being aliased may *not* necessarily refer to an existing file or resource.

Given a defined alias, you may derive a new alias (without the need of calling [[Yii::setAlias()]]) by appending a slash `/` followed with one or more path segments. The aliases defined via [[Yii::setAlias()]] becomes the *root alias*, while aliases derived from it are *derived aliases*. For example, `@foo` is a root alias, while `@foo/bar/file.php` is a derived alias.

You can define an alias using another alias (either root or derived):

```
Yii::setAlias('@foobar', '@foo/bar');
```

Root aliases are usually defined during the bootstrapping stage. For example, you may call [[Yii::setAlias()]] in the entry script. For convenience, Application provides a writable property named `aliases` that you can configure in the application configuration:

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
        '@bar' => 'http://www.example.com',
    ],
];
```

Resolving Aliases

You can call [[Yii::getAlias()]] to resolve a root alias into the file path or URL it represents. The same method can also resolve a derived alias into the corresponding file path or URL:

```
echo Yii::getAlias('@foo');              // displays: /path/to/foo
echo Yii::getAlias('@bar');              // displays: http://www.example.com
echo Yii::getAlias('@foo/bar/file.php'); // displays: /path/to/foo/bar/file.php
```

The path/URL represented by a derived alias is determined by replacing the root alias part with its corresponding path/URL in the derived alias.

> Note: The [[Yii::getAlias()]] method does not check whether the resulting path/URL refers to an existing file or resource.

A root alias may also contain slash `/` characters. The [[Yii::getAlias()]] method is intelligent enough to tell which part of an alias is a root alias and thus correctly determines the corresponding file path or URL:

```
Yii::setAlias('@foo', '/path/to/foo');
Yii::setAlias('@foo/bar', '/path2/bar');
Yii::getAlias('@foo/test/file.php');  // displays: /path/to/foo/test/file.php
Yii::getAlias('@foo/bar/file.php');   // displays: /path2/bar/file.php
```

If `@foo/bar` is not defined as a root alias, the last statement would display `/path/to/foo/bar/file.php`.

Using Aliases

Aliases are recognized in many places in Yii without needing to call [[Yii::getAlias()]] to convert them into paths or URLs. For example, [[yii\caching\FileCache::cachePath]] can accept both a file path and an alias representing a file path, thanks to the `@` prefix which allows it to differentiate a file path from an alias.

```
use yii\caching\FileCache;

$cache = new FileCache([
    'cachePath' => '@runtime/cache',
]);
```

Please pay attention to the API documentation to see if a property or method parameter supports aliases.

Predefined Aliases

Yii predefines a set of aliases to easily reference commonly used file paths and URLs:

- `@yii`, the directory where the `BaseYii.php` file is located (also called the framework directory).
- `@app`, the [[yii\base\Application::basePath|base path]] of the currently running application.
- `@runtime`, the [[yii\base\Application::runtimePath|runtime path]] of the currently running application. Defaults to `@app/runtime`.
- `@webroot`, the Web root directory of the currently running Web application. It is determined based on the directory containing the entry script.
- `@web`, the base URL of the currently running Web application. It has the same value as [[yii\web\Request::baseUrl]].
- `@vendor`, the [[yii\base\Application::vendorPath|Composer vendor directory]]. Defaults to `@app/vendor`.
- `@bower`, the root directory that contains bower packages. Defaults to `@vendor/bower`.
- `@npm`, the root directory that contains npm packages. Defaults to `@vendor/npm`.

The `@yii` alias is defined when you include the `Yii.php` file in your entry script. The rest of the aliases are defined in the application constructor when applying the application configuration.

Extension Aliases

An alias is automatically defined for each extension that is installed via Composer. Each alias is named after the root namespace of the extension as declared in its `composer.json` file, and each alias represents the root directory of the package. For example, if you install the `yiisoft/yii2-jui` extension, you will automatically have the alias `@yii/jui` defined during the bootstrapping stage, equivalent to:

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

Class Autoloading

Yii relies on the class autoloading mechanism to locate and include all required class files. It provides a high-performance class autoloader that is compliant with the PSR-4 standard. The autoloader is installed when you include the `Yii.php` file.

> Note: For simplicity of description, in this section we will only talk about autoloading of classes. However, keep in mind that the content we are describing here applies to autoloading of interfaces and traits as well.

Using the Yii Autoloader

To make use of the Yii class autoloader, you should follow two simple rules when creating and naming your classes:

- Each class must be under a namespace (e.g. `foo\bar\MyClass`)
- Each class must be saved in an individual file whose path is determined by the following algorithm:

```
// $className is a fully qualified class name without the leading backslash
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) . '.php');
```

For example, if a class name and namespace is `foo\bar\MyClass`, the alias for the corresponding class file path would be `@foo/bar/MyClass.php`. In order for this alias to be resolvable into a file path, either `@foo` or `@foo/bar` must be a root alias.

When using the Basic Project Template, you may put your classes under the top-level namespace `app` so that they can be autoloaded by Yii without the need of defining a new alias. This is because `@app` is a predefined alias, and a class name like `app\components\MyClass` can be resolved into the class file `AppBasePath/components/MyClass.php`, according to the algorithm just described.

In the Advanced Project Template, each tier has its own root alias. For example, the front-end tier has a root alias `@frontend`, while the back-end tier root alias is `@backend`. As a result, you may put the front-end classes under the namespace `frontend` while the back-end classes are under `backend`. This will allow these classes to be autoloaded by the Yii autoloader.

Class Map

The Yii class autoloader supports the *class map* feature, which maps class names to the corresponding class file paths. When the autoloader is loading a class, it will first check if the class is found in the map. If so, the corresponding file path will be included directly without further checks. This makes class autoloading super fast. In fact, all core Yii classes are autoloaded this way.

You may add a class to the class map, stored in `Yii::$classMap`, using:

```
Yii::$classMap['foo\bar\MyClass'] = 'path/to/MyClass.php';
```

Aliases can be used to specify class file paths. You should set the class map in the bootstrapping process so that the map is ready before your classes are used.

Using Other Autoloaders

Because Yii embraces Composer as a package dependency manager, it is recommended that you also install the Composer autoloader. If you are using 3rd-party libraries that have their own autoloaders, you should also install those.

When using the Yii autoloader together with other autoloaders, you should include the `Yii.php` file *after* all other autoloaders are installed. This will make the Yii autoloader the first one responding to any class autoloading request. For example, the following code is extracted from the entry script of the Basic Project Template. The first line installs the Composer autoloader, while the second line installs the Yii autoloader:

```
require(__DIR__ . '/../vendor/autoload.php');
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
```

You may use the Composer autoloader alone without the Yii autoloader. However, by doing so, the performance of your class autoloading may be degraded, and you must follow the rules set by Composer in order for your classes to be autoloadable.

> Info: If you do not want to use the Yii autoloader, you must create your own version of the `Yii.php` file and include it in your entry script.

Autoloading Extension Classes

The Yii autoloader is capable of autoloading extension classes. The sole requirement is that an extension specifies the `autoload` section correctly in its `composer.json` file. Please refer to the Composer documentation for more details about specifying `autoload`.

In case you do not use the Yii autoloader, the Composer autoloader can still autoload extension classes for you.

Service Locator

A service locator is an object that knows how to provide all sorts of services (or components) that an application might need. Within a service locator, each component exists as only a single instance, uniquely identified by an ID. You use the ID to retrieve a component from the service locator.

In Yii, a service locator is simply an instance of [[yii\di\ServiceLocator]] or a child class.

The most commonly used service locator in Yii is the *application* object, which can be accessed through `\Yii::$app`. The services it provides are called *application components*, such as the `request`, `response`, and `urlManager` components. You may configure these components, or even replace them with your own implementations, easily through functionality provided by the service locator.

Besides the application object, each module object is also a service locator.

To use a service locator, the first step is to register components with it. A component can be registered via [[yii\di\ServiceLocator::set()]]. The following code shows different ways of registering components:

```
use yii\di\ServiceLocator;
use yii\caching\FileCache;

$locator = new ServiceLocator;

// register "cache" using a class name that can be used to create a component
$locator->set('cache', 'yii\caching\ApcCache');

// register "db" using a configuration array that can be used to create a component
$locator->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=localhost;dbname=demo',
    'username' => 'root',
    'password' => '',
]);

// register "search" using an anonymous function that builds a component
$locator->set('search', function () {
    return new app\components\SolrService;
});

// register "pageCache" using a component
$locator->set('pageCache', new FileCache);
```

Once a component has been registered, you can access it using its ID, in one of the two following ways:

```
$cache = $locator->get('cache');
// or alternatively
$cache = $locator->cache;
```

As shown above, [[yii\di\ServiceLocator]] allows you to access a component like a property using the component ID. When you access a component for the first time, [[yii\di\ServiceLocator]] will use the component registration information to create a new instance of the component and return it. Later, if the component is accessed again, the service locator will return the same instance.

You may use [[yii\di\ServiceLocator::has()]] to check if a component ID has already been registered. If you call [[yii\di\ServiceLocator::get()]] with an invalid ID, an exception will be thrown.

Because service locators are often being created with configurations, a writable property named [[yii\di\ServiceLocator::setComponents()|components]] is provided. This allows you to configure and register multiple components at once. The following code shows a configuration array that can be used to configure a service locator (e.g. an application) with the `db`, `cache` and `search` components:

```
return [
    // ...
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=demo',
            'username' => 'root',
            'password' => '',
        ],
        'cache' => 'yii\caching\ApcCache',
        'search' => function () {
            $solr = new app\components\SolrService('127.0.0.1');
            // ... other initializations ...
            return $solr;
        },
    ],
];
```

In the above, there is an alternative way to configure the `search` component. Instead of directly writing a PHP callback which builds a `SolrService` instance, you can use a static class method to return such a callback, like shown as below:

```
class SolrServiceBuilder
{
    public static function build($ip)
    {
        return function () use ($ip) {
            $solr = new app\components\SolrService($ip);
            // ... other initializations ...
            return $solr;
        };
    }
}

return [
    // ...
    'components' => [
        // ...
        'search' => SolrServiceBuilder::build('127.0.0.1'),
    ],
];
```

This alternative approach is most preferable when you are releasing a Yii component which encapsulates some non-Yii 3rd-party library. You use the static method like shown above to represent the complex logic of building the 3rd-party object, and the user of your component only needs to call the static method to configure the component.

Dependency Injection Container

A dependency injection (DI) container is an object that knows how to instantiate and configure objects and all their dependent objects. Martin Fowler's article has well explained why DI container is useful. Here we will mainly explain the usage of the DI container provided by Yii.

Dependency Injection

Yii provides the DI container feature through the class [[yii\di\Container]]. It supports the following kinds of dependency injection:

- Constructor injection;
- Method injection;
- Setter and property injection;
- PHP callable injection;

Constructor Injection

The DI container supports constructor injection with the help of type hints for constructor parameters. The type hints tell the container which classes or interfaces are dependent when it is used to create a new object. The container will try to get the instances of the dependent classes or interfaces and then inject them into the new object through the constructor. For example,

```
class Foo
{
    public function __construct(Bar $bar)
    {
    }
}

$foo = $container->get('Foo');
// which is equivalent to the following:
$bar = new Bar;
$foo = new Foo($bar);
```

Method Injection

Usually the dependencies of a class are passed to the constructor and are available inside of the class during the whole lifecycle. With Method Injection it is possible to provide a dependency that is only needed by a single method of the class and passing it to the constructor may not be possible or may cause too much overhead in the majority of use cases.

A class method can be defined like the `doSomething()` method in the following example:

```
class MyClass extends \yii\base\Component
{
    public function __construct(/*Some lightweight dependencies here*/, $config = [])
    {
        // ...
    }

    public function doSomething($param1, \my\heavy\Dependency $something)
    {
        // do something with $something
    }
}
```

You may call that method either by passing an instance of `\my\heavy\Dependency` yourself or using [[yii\di\Container::invoke()]] like the following:

```
$obj = new MyClass(/*...*/);
Yii::$container->invoke([$obj, 'doSomething'], ['param1' => 42]); // $something will b
e provided by the DI container
```

Setter and Property Injection

Setter and property injection is supported through configurations. When registering a dependency or when creating a new object, you can provide a configuration which will be used by the container to inject the dependencies through the corresponding setters or properties. For example,

```
use yii\base\Object;

class Foo extends Object
{
    public $bar;

    private $_qux;

    public function getQux()
    {
        return $this->_qux;
    }

    public function setQux(Qux $qux)
    {
        $this->_qux = $qux;
    }
}

$container->get('Foo', [], [
    'bar' => $container->get('Bar'),
    'qux' => $container->get('Qux'),
]);
```

> Info: The [[yii\di\Container::get()]] method takes its third parameter as a configuration array that should be applied to the object being created. If the class implements the [[yii\base\Configurable]] interface (e.g. [[yii\base\Object]]), the configuration array will be passed as the last parameter to the class constructor; otherwise, the configuration will be applied *after* the object is created.

PHP Callable Injection

In this case, the container will use a registered PHP callable to build new instances of a class. Each time when [[yii\di\Container::get()]] is called, the corresponding callable will be invoked. The callable is responsible to resolve the dependencies and inject them appropriately to the newly created objects. For example,

```
$container->set('Foo', function () {
    $foo = new Foo(new Bar);
    // ... other initializations ...
    return $foo;
});

$foo = $container->get('Foo');
```

To hide the complex logic for building a new object, you may use a static class method as callable. For example,

```
class FooBuilder
{
    public static function build()
    {
        $foo = new Foo(new Bar);
        // ... other initializations ...
        return $foo;
    }
}

$container->set('Foo', ['app\helper\FooBuilder', 'build']);

$foo = $container->get('Foo');
```

By doing so, the person who wants to configure the `Foo` class no longer needs to be aware of how it is built.

Registering Dependencies

You can use [[yii\di\Container::set()]] to register dependencies. The registration requires a dependency name as well as a dependency definition. A dependency name can be a class name, an interface name, or an alias name; and a dependency definition can be a class name, a configuration array, or a PHP callable.

```
$container = new \yii\di\Container;

// register a class name as is. This can be skipped.
$container->set('yii\db\Connection');

// register an interface
// When a class depends on the interface, the corresponding class
// will be instantiated as the dependent object
$container->set('yii\mail\MailInterface', 'yii\swiftmailer\Mailer');

// register an alias name. You can use $container->get('foo')
// to create an instance of Connection
$container->set('foo', 'yii\db\Connection');

// register a class with configuration. The configuration
// will be applied when the class is instantiated by get()
$container->set('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// register an alias name with class configuration
// In this case, a "class" element is required to specify the class
$container->set('db', [
    'class' => 'yii\db\Connection',
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);

// register a PHP callable
// The callable will be executed each time when $container->get('db') is called
$container->set('db', function ($container, $params, $config) {
    return new \yii\db\Connection($config);
});

// register a component instance
// $container->get('pageCache') will return the same instance each time it is called
$container->set('pageCache', new FileCache);
```

> Tip: If a dependency name is the same as the corresponding dependency definition, you do not need to register it with the DI container.

A dependency registered via `set()` will generate an instance each time the dependency is needed. You can use [[yii\di\Container::setSingleton()]] to register a dependency that only generates a single instance:

```
$container->setSingleton('yii\db\Connection', [
    'dsn' => 'mysql:host=127.0.0.1;dbname=demo',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

Resolving Dependencies

Once you have registered dependencies, you can use the DI container to create new objects, and the container will automatically resolve dependencies by instantiating them and injecting them into the newly created objects. The dependency resolution is recursive, meaning that if a dependency has other dependencies, those dependencies will also be resolved automatically.

You can use [[yii\di\Container::get()|get()]] to either create or get object instance. The method takes a dependency name, which can be a class name, an interface name or an alias name. The dependency name may be registered via [[yii\di\Container::set()|set()]] or [[yii\di\Container::setSingleton()|setSingleton()]]. You may optionally provide a list of class constructor parameters and a configuration to configure the newly created object.

For example:

```
// "db" is a previously registered alias name
$db = $container->get('db');

// equivalent to: $engine = new \app\components\SearchEngine($apiKey, $apiSecret, ['ty
pe' => 1]);
$engine = $container->get('app\components\SearchEngine', [$apiKey, $apiSecret], ['type'
 => 1]);
```

Behind the scene, the DI container does much more work than just creating a new object. The container will first inspect the class constructor to find out dependent class or interface names and then automatically resolve those dependencies recursively.

The following code shows a more sophisticated example. The `UserLister` class depends on an object implementing the `UserFinderInterface` interface; the `UserFinder` class implements this interface and depends on a `Connection` object. All these dependencies are declared through type hinting of the class constructor parameters. With property dependency registration, the DI container is able to resolve these dependencies automatically and creates a new `UserLister` instance with a simple call of `get('userLister')`.

```php
namespace app\models;

use yii\base\Object;
use yii\db\Connection;
use yii\di\Container;

interface UserFinderInterface
{
    function findUser();
}

class UserFinder extends Object implements UserFinderInterface
{
    public $db;

    public function __construct(Connection $db, $config = [])
    {
        $this->db = $db;
        parent::__construct($config);
    }

    public function findUser()
    {
    }
}

class UserLister extends Object
{
    public $finder;

    public function __construct(UserFinderInterface $finder, $config = [])
    {
        $this->finder = $finder;
        parent::__construct($config);
    }
}

$container = new Container;
$container->set('yii\db\Connection', [
    'dsn' => '...',
]);
$container->set('app\models\UserFinderInterface', [
    'class' => 'app\models\UserFinder',
]);
$container->set('userLister', 'app\models\UserLister');

$lister = $container->get('userLister');

// which is equivalent to:

$db = new \yii\db\Connection(['dsn' => '...']);
$finder = new UserFinder($db);
```

```
$lister = new UserLister($finder);
```

Practical Usage

Yii creates a DI container when you include the `Yii.php` file in the entry script of your application. The DI container is accessible via [[Yii::$container]]. When you call [[Yii::createObject()]], the method will actually call the container's [[yii\di\Container::get()|get()]] method to create a new object. As aforementioned, the DI container will automatically resolve the dependencies (if any) and inject them into obtained object. Because Yii uses [[Yii::createObject()]] in most of its core code to create new objects, this means you can customize the objects globally by dealing with [[Yii::$container]].

For example, let's customize globally the default number of pagination buttons of [[yii\widgets\LinkPager]].

```
\Yii::$container->set('yii\widgets\LinkPager', ['maxButtonCount' => 5]);
```

Now if you use the widget in a view with the following code, the `maxButtonCount` property will be initialized as 5 instead of the default value 10 as defined in the class.

```
echo \yii\widgets\LinkPager::widget();
```

You can still override the value set via DI container, though:

```
echo \yii\widgets\LinkPager::widget(['maxButtonCount' => 20]);
```

> Note: Properties given in the widget call will always override the definition in the DI container. Even if you specify an array, e.g. `'options' => ['id' => 'mypager']` these will not be merged with other options but replace them.

Another example is to take advantage of the automatic constructor injection of the DI container. Assume your controller class depends on some other objects, such as a hotel booking service. You can declare the dependency through a constructor parameter and let the DI container to resolve it for you.

```
namespace app\controllers;

use yii\web\Controller;
use app\components\BookingInterface;

class HotelController extends Controller
{
    protected $bookingService;

    public function __construct($id, $module, BookingInterface $bookingService, $confi
g = [])
    {
        $this->bookingService = $bookingService;
        parent::__construct($id, $module, $config);
    }
}
```

If you access this controller from browser, you will see an error complaining the `BookingInterface` cannot be instantiated. This is because you need to tell the DI container how to deal with this dependency:

```
\Yii::$container->set('app\components\BookingInterface', 'app\components\BookingServic
e');
```

Now if you access the controller again, an instance of `app\components\BookingService` will be created and injected as the 3rd parameter to the controller's constructor.

Advanced Practical Usage

Say we work on API application and have:

- `app\components\Request` class that extends `yii\web\Request` and provides additional functionality
- `app\components\Response` class that extends `yii\web\Response` and should have `format` property set to `json` on creation
- `app\storage\FileStorage` and `app\storage\DocumentsReader` classes that implement some logic on working with documents that are located in some file storage:

```
class FileStorage
{
    public function __contruct($root) {
        // whatever
    }
}

class DocumentsReader
{
    public function __contruct(FileStorage $fs) {
        // whatever
    }
}
```

It is possible to configure multiple definitions at once, passing configuration array to [[yii\di\Container::setDefinitions()|setDefinitions()]] or [[yii\di\Container::setSingletons()|setSingletons()]] method. Iterating over the configuration array, the methods will call [[yii\di\Container::set()|set()]] or [[yii\di\Container::setSingleton()|setSingleton()]] respectively for each item.

The configuration array format is:

- `key` : class name, interface name or alias name. The key will be passed to the [[yii\di\Container::set()|set()]] method as a first argument `$class` .
- `value` : the definition associated with `$class` . Possible values are described in [[yii\di\Container::set()|set()]] documentation for the `$definition` parameter. Will be passed to the [[set()]] method as the second argument `$definition` .

For example, let's configure our container to follow the aforementioned requirements:

```
$container->setDefinitions([
    'yii\web\Request' => 'app\components\Request',
    'yii\web\Response' => [
        'class' => 'app\components\Response',
        'format' => 'json'
    ],
    'app\storage\DocumentsReader' => function () {
        $fs = new app\storage\FileStorage('/var/tempfiles');
        return new app\storage\DocumentsReader($fs);
    }
]);

$reader = $container->get('app\storage\DocumentsReader);
// Will create DocumentReader object with its dependencies as described in the config
```

> Tip: Container may be configured in declarative style using application configuration since version 2.0.11. Check out the Application Configurations subsection of the Configurations guide article.

Everything works, but in case we need to create `DocumentWriter` class, we shall copy-paste the line that creates `FileStorage` object, that is not the smartest way, obviously.

As described in the Resolving Dependencies subsection, [[yii\di\Container::set()|set()]] and [[yii\di\Container::setSingleton()|setSingleton()]] can optionally take dependency's constructor parameters as a third argument. To set the constructor parameters, you may use the following configuration array format:

- `key` : class name, interface name or alias name. The key will be passed to the [[yii\di\Container::set()|set()]] method as a first argument `$class` .
- `value` : array of two elements. The first element will be passed to the [[yii\di\Container::set()|set()]] method as the second argument `$definition` , the second one — as `$params` .

Let's modify our example:

```
$container->setDefinitions([
    'tempFileStorage' => [ // we've created an alias for convenience
        ['class' => 'app\storage\FileStorage'],
        ['/var/tempfiles'] // could be extracted from some config files
    ],
    'app\storage\DocumentsReader' => [
        ['class' => 'app\storage\DocumentsReader'],
        [Instance::of('tempFileStorage')]
    ],
    'app\storage\DocumentsWriter' => [
        ['class' => 'app\storage\DocumentsWriter'],
        [Instance::of('tempFileStorage')]
    ]
]);

$reader = $container->get('app\storage\DocumentsReader);
// Will behave exactly the same as in the previous example.
```

You might notice `Instance::of('tempFileStorage')` notation. It means, that the [[yii\di\Container|Container]] will implicitly provide a dependency registered with the name of `tempFileStorage` and pass it as the first argument of `app\storage\DocumentsWriter` constructor.

> Note: [[yii\di\Container::setDefinitions()|setDefinitions()]] and [[yii\di\Container::setSingletons()|setSingletons()]] methods are available since version 2.0.11.

Another step on configuration optimization is to register some dependencies as singletons. A dependency registered via [[yii\di\Container::set()|set()]] will be instantiated each time it is needed. Some classes do not change the state during runtime, therefore they may be registered as singletons in order to increase the application performance.

A good example could be `app\storage\FileStorage` class, that executes some operations on file system with a simple API (e.g. `$fs->read()`, `$fs->write()`). These operations do not change the internal class state, so we can create its instance once and use it multiple times.

```
$container->setSingletons([
    'tempFileStorage' => [
        ['class' => 'app\storage\FileStorage'],
        ['/var/tempfiles']
    ],
]);

$container->setDefinitions([
    'app\storage\DocumentsReader' => [
        ['class' => 'app\storage\DocumentsReader'],
        [Instance::of('tempFileStorage')]
    ],
    'app\storage\DocumentsWriter' => [
        ['class' => 'app\storage\DocumentsWriter'],
        [Instance::of('tempFileStorage')]
    ]
]);

$reader = $container->get('app\storage\DocumentsReader);
```

When to Register Dependencies

Because dependencies are needed when new objects are being created, their registration should be done as early as possible. The following are the recommended practices:

- If you are the developer of an application, you can register your dependencies using application configuration. Please, read the Application Configurations subsection of the Configurations guide article.
- If you are the developer of a redistributable extension, you can register dependencies in the bootstrapping class of the extension.

Summary

Both dependency injection and service locator are popular design patterns that allow building software in a loosely-coupled and more testable fashion. We highly recommend you to read Martin's article to get a deeper understanding of dependency injection and service locator.

Yii implements its service locator on top of the dependency injection (DI) container. When a service locator is trying to create a new object instance, it will forward the call to the DI container. The latter will resolve the dependencies automatically as described above.

Database Access Objects

Built on top of PDO, Yii DAO (Database Access Objects) provides an object-oriented API for accessing relational databases. It is the foundation for other more advanced database access methods, including query builder and active record.

When using Yii DAO, you mainly need to deal with plain SQLs and PHP arrays. As a result, it is the most efficient way to access databases. However, because SQL syntax may vary for different databases, using Yii DAO also means you have to take extra effort to create a database-agnostic application.

Yii DAO supports the following databases out of box:

- MySQL
- MariaDB
- SQLite
- PostgreSQL
- CUBRID: version 9.3 or higher.
- Oracle
- MSSQL: version 2008 or higher.

> Note: New version of pdo_oci for PHP 7 currently exists only as the source code. Follow instruction provided by community to compile it.

Creating DB Connections

To access a database, you first need to connect to it by creating an instance of [[yii\db\Connection]]:

```
$db = new yii\db\Connection([
    'dsn' => 'mysql:host=localhost;dbname=example',
    'username' => 'root',
    'password' => '',
    'charset' => 'utf8',
]);
```

Because a DB connection often needs to be accessed in different places, a common practice is to configure it in terms of an application component like the following:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=example',
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
    // ...
];
```

You can then access the DB connection via the expression `Yii::$app->db`.

> Tip: You can configure multiple DB application components if your application needs to access multiple databases.

When configuring a DB connection, you should always specify its Data Source Name (DSN) via the [[yii\db\Connection::dsn|dsn]] property. The format of DSN varies for different databases. Please refer to the PHP manual for more details. Below are some examples:

- MySQL, MariaDB: `mysql:host=localhost;dbname=mydatabase`
- SQLite: `sqlite:/path/to/database/file`
- PostgreSQL: `pgsql:host=localhost;port=5432;dbname=mydatabase`
- CUBRID: `cubrid:dbname=demodb;host=localhost;port=33000`
- MS SQL Server (via sqlsrv driver): `sqlsrv:Server=localhost;Database=mydatabase`
- MS SQL Server (via dblib driver): `dblib:host=localhost;dbname=mydatabase`
- MS SQL Server (via mssql driver): `mssql:host=localhost;dbname=mydatabase`
- Oracle: `oci:dbname=//localhost:1521/mydatabase`

Note that if you are connecting with a database via ODBC, you should configure the [[yii\db\Connection::driverName]] property so that Yii can know the actual database type. For example,

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

Besides the [[yii\db\Connection::dsn|dsn]] property, you often need to configure [[yii\db\Connection::username|username]] and [[yii\db\Connection::password|password]]. Please refer to [[yii\db\Connection]] for the full list of configurable properties.

> Info: When you create a DB connection instance, the actual connection to the database is not established until you execute the first SQL or you call the [[yii\db\Connection::open()|open()]] method explicitly.
>
> Tip: Sometimes you may want to execute some queries right after the database connection is established to initialize some environment variables (e.g., to set the timezone or character set). You can do so by registering an event handler for the [[yii\db\Connection::EVENT_AFTER_OPEN|afterOpen]] event of the database connection. You may register the handler directly in the application configuration like so:
>
> ```
> 'db' => [
>     // ...
>     'on afterOpen' => function($event) {
>         // $event->sender refers to the DB connection
>         $event->sender->createCommand("SET time_zone = 'UTC'")->execute();
>     }
> ],
> ```

Executing SQL Queries

Once you have a database connection instance, you can execute a SQL query by taking the following steps:

1. Create a [[yii\db\Command]] with a plain SQL query;
2. Bind parameters (optional);
3. Call one of the SQL execution methods in [[yii\db\Command]].

The following example shows various ways of fetching data from a database:

```
// return a set of rows. each row is an associative array of column names and values.
// an empty array is returned if the query returned no results
$posts = Yii::$app->db->createCommand('SELECT * FROM post')
            ->queryAll();

// return a single row (the first row)
// false is returned if the query has no result
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=1')
           ->queryOne();

// return a single column (the first column)
// an empty array is returned if the query returned no results
$titles = Yii::$app->db->createCommand('SELECT title FROM post')
             ->queryColumn();

// return a scalar value
// false is returned if the query has no result
$count = Yii::$app->db->createCommand('SELECT COUNT(*) FROM post')
             ->queryScalar();
```

> Note: To preserve precision, the data fetched from databases are all represented as strings, even if the corresponding database column types are numerical.

Binding Parameters

When creating a DB command from a SQL with parameters, you should almost always use the approach of binding parameters to prevent SQL injection attacks. For example,

```
$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND status=:stat
us')
           ->bindValue(':id', $_GET['id'])
           ->bindValue(':status', 1)
           ->queryOne();
```

In the SQL statement, you can embed one or multiple parameter placeholders (e.g. `:id` in the above example). A parameter placeholder should be a string starting with a colon. You may then call one of the following parameter binding methods to bind the parameter values:

- [[yii\db\Command::bindValue()|bindValue()]]: bind a single parameter value
- [[yii\db\Command::bindValues()|bindValues()]]: bind multiple parameter values in one call
- [[yii\db\Command::bindParam()|bindParam()]]: similar to [[yii\db\Command::bindValue()|bindValue()]] but also support binding parameter references.

The following example shows alternative ways of binding parameters:

```
$params = [':id' => $_GET['id'], ':status' => 1];

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND status=:stat
us')
            ->bindValues($params)
            ->queryOne();

$post = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id AND status=:stat
us', $params)
            ->queryOne();
```

Parameter binding is implemented via prepared statements. Besides preventing SQL injection attacks, it may also improve performance by preparing a SQL statement once and executing it multiple times with different parameters. For example,

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id');

$post1 = $command->bindValue(':id', 1)->queryOne();
$post2 = $command->bindValue(':id', 2)->queryOne();
// ...
```

Because [[yii\db\Command::bindParam()|bindParam()]] supports binding parameters by references, the above code can also be written like the following:

```
$command = Yii::$app->db->createCommand('SELECT * FROM post WHERE id=:id')
              ->bindParam(':id', $id);

$id = 1;
$post1 = $command->queryOne();

$id = 2;
$post2 = $command->queryOne();
// ...
```

Notice that you bind the placeholder to the `$id` variable before the execution, and then change the value of that variable before each subsequent execution (this is often done with loops). Executing queries in this manner can be vastly more efficient than running a new query for every different parameter value.

Executing Non-SELECT Queries

The `queryXyz()` methods introduced in the previous sections all deal with SELECT queries which fetch data from databases. For queries that do not bring back data, you should call the [[yii\db\Command::execute()]] method instead. For example,

```
Yii::$app->db->createCommand('UPDATE post SET status=1 WHERE id=1')
   ->execute();
```

The [[yii\db\Command::execute()]] method returns the number of rows affected by the SQL execution.

For INSERT, UPDATE and DELETE queries, instead of writing plain SQLs, you may call [[yii\db\Command::insert()|insert()]], [[yii\db\Command::update()|update()]], [[yii\db\Command::delete()|delete()]], respectively, to build the corresponding SQLs. These methods will properly quote table and column names and bind parameter values. For example,

```
// INSERT (table name, column values)
Yii::$app->db->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
])->execute();

// UPDATE (table name, column values, condition)
Yii::$app->db->createCommand()->update('user', ['status' => 1], 'age > 30')->execute()
;

// DELETE (table name, condition)
Yii::$app->db->createCommand()->delete('user', 'status = 0')->execute();
```

You may also call [[yii\db\Command::batchInsert()|batchInsert()]] to insert multiple rows in one shot, which is much more efficient than inserting one row at a time:

```
// table name, column names, column values
Yii::$app->db->createCommand()->batchInsert('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
])->execute();
```

Note that the aforementioned methods only create the query and you always have to call [[yii\db\Command::execute()|execute()]] to actually run them.

Quoting Table and Column Names

When writing database-agnostic code, properly quoting table and column names is often a headache because different databases have different name quoting rules. To overcome this problem, you may use the following quoting syntax introduced by Yii:

- `[[column name]]`: enclose a column name to be quoted in double square brackets;
- `{{table name}}`: enclose a table name to be quoted in double curly brackets.

Yii DAO will automatically convert such constructs into the corresponding quoted column or table names using the DBMS specific syntax. For example,

```
// executes this SQL for MySQL: SELECT COUNT(`id`) FROM `employee`
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM {{employee}}")
            ->queryScalar();
```

Using Table Prefix

If most of your DB tables names share a common prefix, you may use the table prefix feature provided by Yii DAO.

First, specify the table prefix via the [[yii\db\Connection::tablePrefix]] property in the application config:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

Then in your code, whenever you need to refer to a table whose name contains such a prefix, use the syntax `{{%table_name}}`. The percentage character will be automatically replaced with the table prefix that you have specified when configuring the DB connection. For example,

```
// executes this SQL for MySQL: SELECT COUNT(`id`) FROM `tbl_employee`
$count = Yii::$app->db->createCommand("SELECT COUNT([[id]]) FROM {{%employee}}")
            ->queryScalar();
```

Performing Transactions

When running multiple related queries in a sequence, you may need to wrap them in a transaction to ensure the integrity and consistency of your database. If any of the queries fails, the database will be rolled back to the state as if none of these queries were executed.

The following code shows a typical way of using transactions:

```
Yii::$app->db->transaction(function($db) {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... executing other SQL statements ...
});
```

The above code is equivalent to the following, which gives you more control about the error handling code:

```
$db = Yii::$app->db;
$transaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();
    $db->createCommand($sql2)->execute();
    // ... executing other SQL statements ...

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

By calling the [[yii\db\Connection::beginTransaction()|beginTransaction()]] method, a new transaction is started. The transaction is represented as a [[yii\db\Transaction]] object stored in the `$transaction` variable. Then, the queries being executed are enclosed in a `try...catch...` block. If all queries are executed successfully, the [[yii\db\Transaction::commit()|commit()]] method is called to commit the transaction. Otherwise, if an exception will be triggered and caught, the [[yii\db\Transaction::rollBack()|rollBack()]] method is called to roll back the changes made by the queries prior to that failed query in the transaction. `throw $e` will then re-throw the exception as if we had not caught it, so the normal error handling process will take care of it.

> Note: in the above code we have two catch-blocks for compatibility with PHP 5.x and PHP 7.x. `\Exception` implements the `\Throwable` interface since PHP 7.0, so you can skip the part with `\Exception` if your app uses only PHP 7.0 and higher.

Specifying Isolation Levels

Yii also supports setting isolation levels for your transactions. By default, when starting a new transaction, it will use the default isolation level set by your database system. You can override the default isolation level as follows,

```
$isolationLevel = \yii\db\Transaction::REPEATABLE_READ;

Yii::$app->db->transaction(function ($db) {
    ....
}, $isolationLevel);

// or alternatively

$transaction = Yii::$app->db->beginTransaction($isolationLevel);
```

Yii provides four constants for the most common isolation levels:

- [[\yii\db\Transaction::READ_UNCOMMITTED]] - the weakest level, Dirty reads, non-repeatable reads and phantoms may occur.
- [[\yii\db\Transaction::READ_COMMITTED]] - avoid dirty reads.
- [[\yii\db\Transaction::REPEATABLE_READ]] - avoid dirty reads and non-repeatable reads.
- [[\yii\db\Transaction::SERIALIZABLE]] - the strongest level, avoids all of the above named problems.

Besides using the above constants to specify isolation levels, you may also use strings with a valid syntax supported by the DBMS that you are using. For example, in PostgreSQL, you may use `SERIALIZABLE READ ONLY DEFERRABLE`.

Note that some DBMS allow setting the isolation level only for the whole connection. Any subsequent transactions will get the same isolation level even if you do not specify any. When using this feature you may need to set the isolation level for all transactions explicitly to avoid conflicting settings. At the time of this writing, only MSSQL and SQLite are affected by this limitation.

> Note: SQLite only supports two isolation levels, so you can only use `READ UNCOMMITTED` and `SERIALIZABLE`. Usage of other levels will result in an exception being thrown.
>
> Note: PostgreSQL does not allow setting the isolation level before the transaction starts so you can not specify the isolation level directly when starting the transaction. You have to call [[yii\db\Transaction::setIsolationLevel()]] in this case after the transaction has started.

Nesting Transactions

If your DBMS supports Savepoint, you may nest multiple transactions like the following:

```
Yii::$app->db->transaction(function ($db) {
    // outer transaction

    $db->transaction(function ($db) {
        // inner transaction
    });
});
```

Or alternatively,

```
$db = Yii::$app->db;
$outerTransaction = $db->beginTransaction();
try {
    $db->createCommand($sql1)->execute();

    $innerTransaction = $db->beginTransaction();
    try {
        $db->createCommand($sql2)->execute();
        $innerTransaction->commit();
    } catch (\Exception $e) {
        $innerTransaction->rollBack();
        throw $e;
    } catch (\Throwable $e) {
        $innerTransaction->rollBack();
        throw $e;
    }

    $outerTransaction->commit();
} catch (\Exception $e) {
    $outerTransaction->rollBack();
    throw $e;
} catch (\Throwable $e) {
    $outerTransaction->rollBack();
    throw $e;
}
```

Replication and Read-Write Splitting

Many DBMS support database replication#Database_replication) to get better database availability and faster server response time. With database replication, data are replicated from the so-called *master servers* to *slave servers*. All writes and updates must take place on the master servers, while reads may also take place on the slave servers.

To take advantage of database replication and achieve read-write splitting, you can configure a [[yii\db\Connection]] component like the following:

```
[
    'class' => 'yii\db\Connection',

    // configuration for the master
    'dsn' => 'dsn for master server',
    'username' => 'master',
    'password' => '',

    // common configuration for slaves
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // list of slave configurations
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]
```

The above configuration specifies a setup with a single master and multiple slaves. One of the slaves will be connected and used to perform read queries, while the master will be used to perform write queries. Such read-write splitting is accomplished automatically with this configuration. For example,

```
// create a Connection instance using the above configuration
Yii::$app->db = Yii::createObject($config);

// query against one of the slaves
$rows = Yii::$app->db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();

// query against the master
Yii::$app->db->createCommand("UPDATE user SET username='demo' WHERE id=1")->execute();
```

> Info: Queries performed by calling [[yii\db\Command::execute()]] are considered as write queries, while all other queries done through one of the "query" methods of [[yii\db\Command]] are read queries. You can get the currently active slave connection via `Yii::$app->db->slave`.

The `Connection` component supports load balancing and failover between slaves. When performing a read query for the first time, the `Connection` component will randomly pick a slave and try connecting to it. If the slave is found "dead", it will try another one. If none of the slaves is available, it will connect to the master. By configuring a [[yii\db\Connection::serverStatusCache|server status cache]], a "dead" server can be remembered so that it will not be tried again during a [[yii\db\Connection::serverRetryInterval|certain period of time]].

> Info: In the above configuration, a connection timeout of 10 seconds is specified for every slave. This means if a slave cannot be reached in 10 seconds, it is considered as "dead". You can adjust this parameter based on your actual environment.

You can also configure multiple masters with multiple slaves. For example,

```
[
    'class' => 'yii\db\Connection',

    // common configuration for masters
    'masterConfig' => [
        'username' => 'master',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // list of master configurations
    'masters' => [
        ['dsn' => 'dsn for master server 1'],
        ['dsn' => 'dsn for master server 2'],
    ],

    // common configuration for slaves
    'slaveConfig' => [
        'username' => 'slave',
        'password' => '',
        'attributes' => [
            // use a smaller connection timeout
            PDO::ATTR_TIMEOUT => 10,
        ],
    ],

    // list of slave configurations
    'slaves' => [
        ['dsn' => 'dsn for slave server 1'],
        ['dsn' => 'dsn for slave server 2'],
        ['dsn' => 'dsn for slave server 3'],
        ['dsn' => 'dsn for slave server 4'],
    ],
]
```

The above configuration specifies two masters and four slaves. The `Connection` component also supports load balancing and failover between masters just as it does between slaves. A difference is that when none of the masters are available an exception will be thrown.

> Note: When you use the [[yii\db\Connection::masters|masters]] property to configure one or multiple masters, all other properties for specifying a database connection (e.g. `dsn`, `username`, `password`) with the `Connection` object itself will be ignored.

By default, transactions use the master connection. And within a transaction, all DB operations will use the master connection. For example,

```
$db = Yii::$app->db;
// the transaction is started on the master connection
$transaction = $db->beginTransaction();

try {
    // both queries are performed against the master
    $rows = $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
    $db->createCommand("UPDATE user SET username='demo' WHERE id=1")->execute();

    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

If you want to start a transaction with the slave connection, you should explicitly do so, like the following:

```
$transaction = Yii::$app->db->slave->beginTransaction();
```

Sometimes, you may want to force using the master connection to perform a read query. This can be achieved with the `useMaster()` method:

```
$rows = Yii::$app->db->useMaster(function ($db) {
    return $db->createCommand('SELECT * FROM user LIMIT 10')->queryAll();
});
```

You may also directly set `Yii::$app->db->enableSlaves` to be `false` to direct all queries to the master connection.

Working with Database Schema

Yii DAO provides a whole set of methods to let you manipulate the database schema, such as creating new tables, dropping a column from a table, etc. These methods are listed as follows:

- [[yii\db\Command::createTable()|createTable()]]: creating a table
- [[yii\db\Command::renameTable()|renameTable()]]: renaming a table
- [[yii\db\Command::dropTable()|dropTable()]]: removing a table
- [[yii\db\Command::truncateTable()|truncateTable()]]: removing all rows in a table

- [[yii\db\Command::addColumn()|addColumn()]]: adding a column
- [[yii\db\Command::renameColumn()|renameColumn()]]: renaming a column
- [[yii\db\Command::dropColumn()|dropColumn()]]: removing a column
- [[yii\db\Command::alterColumn()|alterColumn()]]: altering a column
- [[yii\db\Command::addPrimaryKey()|addPrimaryKey()]]: adding a primary key
- [[yii\db\Command::dropPrimaryKey()|dropPrimaryKey()]]: removing a primary key
- [[yii\db\Command::addForeignKey()|addForeignKey()]]: adding a foreign key
- [[yii\db\Command::dropForeignKey()|dropForeignKey()]]: removing a foreign key
- [[yii\db\Command::createIndex()|createIndex()]]: creating an index
- [[yii\db\Command::dropIndex()|dropIndex()]]: removing an index

These methods can be used like the following:

```
// CREATE TABLE
Yii::$app->db->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

The above array describes the name and types of the columns to be created. For the column types, Yii provides a set of abstract data types, that allow you to define a database agnostic schema. These are converted to DBMS specific type definitions dependent on the database, the table is created in. Please refer to the API documentation of the [[yii\db\Command::createTable()|createTable()]]-method for more information.

Besides changing the database schema, you can also retrieve the definition information about a table through the [[yii\db\Connection::getTableSchema()|getTableSchema()]] method of a DB connection. For example,

```
$table = Yii::$app->db->getTableSchema('post');
```

The method returns a [[yii\db\TableSchema]] object which contains the information about the table's columns, primary keys, foreign keys, etc. All these information are mainly utilized by query builder and active record to help you write database-agnostic code.

Query Builder

Built on top of Database Access Objects, query builder allows you to construct a SQL query in a programmatic and DBMS-agnostic way. Compared to writing raw SQL statements, using query builder will help you write more readable SQL-related code and generate more secure SQL statements.

Using query builder usually involves two steps:

1. Build a [[yii\db\Query]] object to represent different parts (e.g. `SELECT`, `FROM`) of a SELECT SQL statement.
2. Execute a query method (e.g. `all()`) of [[yii\db\Query]] to retrieve data from the database.

The following code shows a typical way of using query builder:

```
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->all();
```

The above code generates and executes the following SQL query, where the `:last_name` parameter is bound with the string `'Smith'`.

```
SELECT `id`, `email`
FROM `user`
WHERE `last_name` = :last_name
LIMIT 10
```

> Info: You usually mainly work with [[yii\db\Query]] instead of [[yii\db\QueryBuilder]]. The latter is invoked by the former implicitly when you call one of the query methods. [[yii\db\QueryBuilder]] is the class responsible for generating DBMS-dependent SQL statements (e.g. quoting table/column names differently) from DBMS-independent [[yii\db\Query]] objects.

Building Queries

To build a [[yii\db\Query]] object, you call different query building methods to specify different parts of a SQL query. The names of these methods resemble the SQL keywords used in the corresponding parts of the SQL statement. For example, to specify the `FROM` part of a SQL query, you would call the [[yii\db\Query::from()|from()]] method. All the query building methods return the query object itself, which allows you to chain multiple calls together.

In the following, we will describe the usage of each query building method.

[[yii\db\Query::select()|select()]]

The [[yii\db\Query::select()|select()]] method specifies the `SELECT` fragment of a SQL statement. You can specify columns to be selected in either an array or a string, like the following. The column names being selected will be automatically quoted when the SQL statement is being generated from a query object.

```
$query->select(['id', 'email']);

// equivalent to:

$query->select('id, email');
```

The column names being selected may include table prefixes and/or column aliases, like you do when writing raw SQL queries. For example,

```
$query->select(['user.id AS user_id', 'email']);

// equivalent to:

$query->select('user.id AS user_id, email');
```

If you are using the array format to specify columns, you can also use the array keys to specify the column aliases. For example, the above code can be rewritten as follows,

```
$query->select(['user_id' => 'user.id', 'email']);
```

If you do not call the [[yii\db\Query::select()|select()]] method when building a query, `*` will be selected, which means selecting *all* columns.

Besides column names, you can also select DB expressions. You must use the array format when selecting a DB expression that contains commas to avoid incorrect automatic name quoting. For example,

```
$query->select(["CONCAT(first_name, ' ', last_name) AS full_name", 'email']);
```

As with all places where raw SQL is involved, you may use the DBMS agnostic quoting syntax for table and column names when writing DB expressions in select.

Starting from version 2.0.1, you may also select sub-queries. You should specify each sub-query in terms of a [[yii\db\Query]] object. For example,

```
$subQuery = (new Query())->select('COUNT(*)')->from('user');

// SELECT `id`, (SELECT COUNT(*) FROM `user`) AS `count` FROM `post`
$query = (new Query())->select(['id', 'count' => $subQuery])->from('post');
```

To select distinct rows, you may call [[yii\db\Query::distinct()|distinct()]], like the following:

```
// SELECT DISTINCT `user_id` ...
$query->select('user_id')->distinct();
```

You can call [[yii\db\Query::addSelect()|addSelect()]] to select additional columns. For example,

```
$query->select(['id', 'username'])
    ->addSelect(['email']);
```

[[yii\db\Query::from()|from()]]

The [[yii\db\Query::from()|from()]] method specifies the `FROM` fragment of a SQL statement. For example,

```
// SELECT * FROM `user`
$query->from('user');
```

You can specify the table(s) being selected from in either a string or an array. The table names may contain schema prefixes and/or table aliases, like you do when writing raw SQL statements. For example,

```
$query->from(['public.user u', 'public.post p']);

// equivalent to:

$query->from('public.user u, public.post p');
```

If you are using the array format, you can also use the array keys to specify the table aliases, like the following:

```
$query->from(['u' => 'public.user', 'p' => 'public.post']);
```

Besides table names, you can also select from sub-queries by specifying them in terms of [[yii\db\Query]] objects. For example,

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');

// SELECT * FROM (SELECT `id` FROM `user` WHERE status=1) u
$query->from(['u' => $subQuery]);
```

Prefixes

Also a default [[yii\db\Connection::$tablePrefix|tablePrefix]] can be applied. Implementation instructions are in the "Quoting Tables" section of the "Database Access Objects" guide.

[[yii\db\Query::where()|where()]]

The [[yii\db\Query::where()|where()]] method specifies the `WHERE` fragment of a SQL query. You can use one of the three formats to specify a `WHERE` condition:

- string format, e.g., `'status=1'`
- hash format, e.g. `['status' => 1, 'type' => 2]`
- operator format, e.g. `['like', 'name', 'test']`

String Format

String format is best used to specify very simple conditions or if you need to use built-in functions of the DBMS. It works as if you are writing a raw SQL. For example,

```
$query->where('status=1');

// or use parameter binding to bind dynamic parameter values
$query->where('status=:status', [':status' => $status]);

// raw SQL using MySQL YEAR() function on a date field
$query->where('YEAR(somedate) = 2015');
```

Do NOT embed variables directly in the condition like the following, especially if the variable values come from end user inputs, because this will make your application subject to SQL injection attacks.

```
// Dangerous! Do NOT do this unless you are very certain $status must be an integer.
$query->where("status=$status");
```

When using parameter binding, you may call [[yii\db\Query::params()|params()]] or [[yii\db\Query::addParams()|addParams()]] to specify parameters separately.

```
$query->where('status=:status')
    ->addParams([':status' => $status]);
```

As with all places where raw SQL is involved, you may use the DBMS agnostic quoting syntax for table and column names when writing conditions in string format.

Hash Format

Hash format is best used to specify multiple `AND`-concatenated sub-conditions each being a simple equality assertion. It is written as an array whose keys are column names and values the corresponding values that the columns should be. For example,

```
// ...WHERE (`status` = 10) AND (`type` IS NULL) AND (`id` IN (4, 8, 15))
$query->where([
    'status' => 10,
    'type' => null,
    'id' => [4, 8, 15],
]);
```

As you can see, the query builder is intelligent enough to properly handle values that are nulls or arrays.

You can also use sub-queries with hash format like the following:

```
$userQuery = (new Query())->select('id')->from('user');

// ...WHERE `id` IN (SELECT `id` FROM `user`)
$query->where(['id' => $userQuery]);
```

Using the Hash Format, Yii internally uses parameter binding so in contrast to the string format, here you do not have to add parameters manually.

Operator Format

Operator format allows you to specify arbitrary conditions in a programmatic way. It takes the following format:

```
[operator, operand1, operand2, ...]
```

where the operands can each be specified in string format, hash format or operator format recursively, while the operator can be one of the following:

- `and` : the operands should be concatenated together using `AND` . For example, `['and', 'id=1', 'id=2']` will generate `id=1 AND id=2` . If an operand is an array, it will be converted into a string using the rules described here. For example, `['and', 'type=1', ['or', 'id=1', 'id=2']]` will generate `type=1 AND (id=1 OR id=2)` . The method will NOT do any quoting or escaping.
- `or` : similar to the `and` operator except that the operands are concatenated using `OR` .
- `between` : operand 1 should be the column name, and operand 2 and 3 should be the starting and ending values of the range that the column is in. For example, `['between', 'id', 1, 10]` will generate `id BETWEEN 1 AND 10` .
- `not between` : similar to `between` except the `BETWEEN` is replaced with `NOT BETWEEN` in the generated condition.
- `in` : operand 1 should be a column or DB expression. Operand 2 can be either an array or a `Query` object. It will generate an `IN` condition. If Operand 2 is an array, it will represent the range of the values that the column or DB expression should be; If Operand 2 is a `Query` object, a sub-query will be generated and used as the range of the column or DB expression. For example, `['in', 'id', [1, 2, 3]]` will generate `id IN (1, 2, 3)` . The method will properly quote the column name and escape values in the range. The `in` operator also supports composite columns. In this case, operand 1 should be an array of the columns, while operand 2 should be an array of arrays or a `Query` object representing the range of the columns.
- `not in` : similar to the `in` operator except that `IN` is replaced with `NOT IN` in the generated condition.
- `like` : operand 1 should be a column or DB expression, and operand 2 be a string or an array representing the values that the column or DB expression should be like. For example, `['like', 'name', 'tester']` will generate `name LIKE '%tester%'` . When the value range is given as an array, multiple `LIKE` predicates will be generated and concatenated using `AND` . For example, `['like', 'name', ['test', 'sample']]` will generate `name LIKE '%test%' AND name LIKE '%sample%'` . You may also provide an

optional third operand to specify how to escape special characters in the values. The operand should be an array of mappings from the special characters to their escaped counterparts. If this operand is not provided, a default escape mapping will be used. You may use `false` or an empty array to indicate the values are already escaped and no escape should be applied. Note that when using an escape mapping (or the third operand is not provided), the values will be automatically enclosed within a pair of percentage characters.

> Note: When using PostgreSQL you may also use `ilike` instead of `like` for case-insensitive matching.

- `or like` : similar to the `like` operator except that `OR` is used to concatenate the `LIKE` predicates when operand 2 is an array.
- `not like` : similar to the `like` operator except that `LIKE` is replaced with `NOT LIKE` in the generated condition.
- `or not like` : similar to the `not like` operator except that `OR` is used to concatenate the `NOT LIKE` predicates.
- `exists` : requires one operand which must be an instance of [[yii\db\Query]] representing the sub-query. It will build an `EXISTS (sub-query)` expression.
- `not exists` : similar to the `exists` operator and builds a `NOT EXISTS (sub-query)` expression.
- `>` , `<=` , or any other valid DB operator that takes two operands: the first operand must be a column name while the second operand a value. For example, `['>', 'age', 10]` will generate `age>10` .

Using the Operator Format, Yii internally uses parameter binding so in contrast to the string format, here you do not have to add parameters manually.

Appending Conditions

You can use [[yii\db\Query::andWhere()|andWhere()]] or [[yii\db\Query::orWhere()|orWhere()]] to append additional conditions to an existing one. You can call them multiple times to append multiple conditions separately. For example,

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);

if (!empty($search)) {
    $query->andWhere(['like', 'title', $search]);
}
```

If `$search` is not empty, the following `WHERE` condition will be generated:

```
WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

Filter Conditions

When building `WHERE` conditions based on input from end users, you usually want to ignore those input values, that are empty. For example, in a search form that allows you to search by username and email, you would like to ignore the username/email condition if the user does not enter anything in the username/email input field. You can achieve this goal by using the [[yii\db\Query::filterWhere()|filterWhere()]] method:

```
// $username and $email are from user inputs
$query->filterWhere([
    'username' => $username,
    'email' => $email,
]);
```

The only difference between [[yii\db\Query::filterWhere()|filterWhere()]] and [[yii\db\Query::where()|where()]] is that the former will ignore empty values provided in the condition in hash format. So if `$email` is empty while `$username` is not, the above code will result in the SQL condition `WHERE username=:username`.

> Info: A value is considered empty if it is `null`, an empty array, an empty string or a string consisting of whitespaces only.

Like [[yii\db\Query::andWhere()|andWhere()]] and [[yii\db\Query::orWhere()|orWhere()]], you can use [[yii\db\Query::andFilterWhere()|andFilterWhere()]] and [[yii\db\Query::orFilterWhere()|orFilterWhere()]] to append additional filter conditions to the existing one.

Additionally, there is [[yii\db\Query::andFilterCompare()]] that can intelligently determine operator based on what's in the value:

```
$query->andFilterCompare('name', 'John Doe');
$query->andFilterCompare('rating', '>9');
$query->andFilterCompare('value', '<=100');
```

You can also specify operator explicitly:

```
$query->andFilterCompare('name', 'Doe', 'like');
```

Since Yii 2.0.11 there are similar methods for `HAVING` condition:

- [[yii\db\Query::filterHaving()|filterHaving()]]
- [[yii\db\Query::andFilterHaving()|andFilterHaving()]]
- [[yii\db\Query::orFilterHaving()|orFilterHaving()]]

[[yii\db\Query::orderBy()|orderBy()]]

The [[yii\db\Query::orderBy()|orderBy()]] method specifies the `ORDER BY` fragment of a SQL query. For example,

```
// ... ORDER BY `id` ASC, `name` DESC
$query->orderBy([
    'id' => SORT_ASC,
    'name' => SORT_DESC,
]);
```

In the above code, the array keys are column names while the array values are the corresponding order by directions. The PHP constant `SORT_ASC` specifies ascending sort and `SORT_DESC` descending sort.

If `ORDER BY` only involves simple column names, you can specify it using a string, just like you do when writing raw SQL statements. For example,

```
$query->orderBy('id ASC, name DESC');
```

> Note: You should use the array format if `ORDER BY` involves some DB expression.

You can call [[yii\db\Query::addOrderBy()|addOrderBy()]] to add additional columns to the `ORDER BY` fragment. For example,

```
$query->orderBy('id ASC')
    ->addOrderBy('name DESC');
```

[[yii\db\Query::groupBy()|groupBy()]]

The [[yii\db\Query::groupBy()|groupBy()]] method specifies the `GROUP BY` fragment of a SQL query. For example,

```
// ... GROUP BY `id`, `status`
$query->groupBy(['id', 'status']);
```

If `GROUP BY` only involves simple column names, you can specify it using a string, just like you do when writing raw SQL statements. For example,

```
$query->groupBy('id, status');
```

> Note: You should use the array format if `GROUP BY` involves some DB expression.

You can call [[yii\db\Query::addGroupBy()|addGroupBy()]] to add additional columns to the `GROUP BY` fragment. For example,

```
$query->groupBy(['id', 'status'])
    ->addGroupBy('age');
```

[[yii\db\Query::having()|having()]]

The [[yii\db\Query::having()|having()]] method specifies the `HAVING` fragment of a SQL query. It takes a condition which can be specified in the same way as that for where(). For example,

```
// ... HAVING `status` = 1
$query->having(['status' => 1]);
```

Please refer to the documentation for where() for more details about how to specify a condition.

You can call [[yii\db\Query::andHaving()|andHaving()]] or [[yii\db\Query::orHaving()|orHaving()]] to append additional conditions to the `HAVING` fragment. For example,

```
// ... HAVING (`status` = 1) AND (`age` > 30)
$query->having(['status' => 1])
    ->andHaving(['>', 'age', 30]);
```

[[yii\db\Query::limit()|limit()]] and [[yii\db\Query::offset()|offset()]]

The [[yii\db\Query::limit()|limit()]] and [[yii\db\Query::offset()|offset()]] methods specify the `LIMIT` and `OFFSET` fragments of a SQL query. For example,

```
// ... LIMIT 10 OFFSET 20
$query->limit(10)->offset(20);
```

If you specify an invalid limit or offset (e.g. a negative value), it will be ignored.

> Info: For DBMS that do not support `LIMIT` and `OFFSET` (e.g. MSSQL), query builder will generate a SQL statement that emulates the `LIMIT` / `OFFSET` behavior.

[[yii\db\Query::join()|join()]]

The [[yii\db\Query::join()|join()]] method specifies the `JOIN` fragment of a SQL query. For example,

```
// ... LEFT JOIN `post` ON `post`.`user_id` = `user`.`id`
$query->join('LEFT JOIN', 'post', 'post.user_id = user.id');
```

The [[yii\db\Query::join()|join()]] method takes four parameters:

- `$type` : join type, e.g., `'INNER JOIN'` , `'LEFT JOIN'` .
- `$table` : the name of the table to be joined.
- `$on` : optional, the join condition, i.e., the `ON` fragment. Please refer to where() for details about specifying a condition. Note, that the array syntax does **not** work for specifying a column based condition, e.g. `['user.id' => 'comment.userId']` will result in a condition where the user id must be equal to the string `'comment.userId'` . You should use the string syntax instead and specify the condition as `'user.id = comment.userId'` .
- `$params` : optional, the parameters to be bound to the join condition.

You can use the following shortcut methods to specify `INNER JOIN` , `LEFT JOIN` and `RIGHT JOIN` , respectively.

- [[yii\db\Query::innerJoin()|innerJoin()]]
- [[yii\db\Query::leftJoin()|leftJoin()]]
- [[yii\db\Query::rightJoin()|rightJoin()]]

For example,

```
$query->leftJoin('post', 'post.user_id = user.id');
```

To join with multiple tables, call the above join methods multiple times, once for each table.

Besides joining with tables, you can also join with sub-queries. To do so, specify the sub-queries to be joined as [[yii\db\Query]] objects. For example,

```
$subQuery = (new \yii\db\Query())->from('post');
$query->leftJoin(['u' => $subQuery], 'u.id = author_id');
```

In this case, you should put the sub-query in an array and use the array key to specify the alias.

[[yii\db\Query::union()|union()]]

The [[yii\db\Query::union()|union()]] method specifies the `UNION` fragment of a SQL query. For example,

```
$query1 = (new \yii\db\Query())
    ->select("id, category_id AS type, name")
    ->from('post')
    ->limit(10);

$query2 = (new \yii\db\Query())
    ->select('id, type, name')
    ->from('user')
    ->limit(10);

$query1->union($query2);
```

You can call [[yii\db\Query::union()|union()]] multiple times to append more `UNION` fragments.

Query Methods

[[yii\db\Query]] provides a whole set of methods for different query purposes:

- [[yii\db\Query::all()|all()]]: returns an array of rows with each row being an associative array of name-value pairs.
- [[yii\db\Query::one()|one()]]: returns the first row of the result.
- [[yii\db\Query::column()|column()]]: returns the first column of the result.
- [[yii\db\Query::scalar()|scalar()]]: returns a scalar value located at the first row and first column of the result.
- [[yii\db\Query::exists()|exists()]]: returns a value indicating whether the query contains any result.

- [[yii\db\Query::count()|count()]]: returns the result of a `COUNT` query.
- Other aggregation query methods, including [[yii\db\Query::sum()|sum($q)]], [[yii\db\Query::average()|average($q)]], [[yii\db\Query::max()|max($q)]], [[yii\db\Query::min()|min($q)]]. The `$q` parameter is mandatory for these methods and can be either a column name or a DB expression.

For example,

```
// SELECT `id`, `email` FROM `user`
$rows = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->all();

// SELECT * FROM `user` WHERE `username` LIKE `%test%`
$row = (new \yii\db\Query())
    ->from('user')
    ->where(['like', 'username', 'test'])
    ->one();
```

> Note: The [[yii\db\Query::one()|one()]] method only returns the first row of the query result. It does NOT add `LIMIT 1` to the generated SQL statement. This is fine and preferred if you know the query will return only one or a few rows of data (e.g. if you are querying with some primary keys). However, if the query may potentially result in many rows of data, you should call `limit(1)` explicitly to improve the performance, e.g., `(new \yii\db\Query())->from('user')->limit(1)->one()`.

All these query methods take an optional `$db` parameter representing the [[yii\db\Connection|DB connection]] that should be used to perform a DB query. If you omit this parameter, the `db` application component will be used as the DB connection. Below is another example using the [[yii\db\Query::count()|count()]] query method:

```
// executes SQL: SELECT COUNT(*) FROM `user` WHERE `last_name`=:last_name
$count = (new \yii\db\Query())
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->count();
```

When you call a query method of [[yii\db\Query]], it actually does the following work internally:

- Call [[yii\db\QueryBuilder]] to generate a SQL statement based on the current construct of [[yii\db\Query]];
- Create a [[yii\db\Command]] object with the generated SQL statement;
- Call a query method (e.g. [[yii\db\Command::queryAll()|queryAll()]]) of

[[yii\db\Command]] to execute the SQL statement and retrieve the data.

Sometimes, you may want to examine or use the SQL statement built from a [[yii\db\Query]] object. You can achieve this goal with the following code:

```
$command = (new \yii\db\Query())
    ->select(['id', 'email'])
    ->from('user')
    ->where(['last_name' => 'Smith'])
    ->limit(10)
    ->createCommand();

// show the SQL statement
echo $command->sql;
// show the parameters to be bound
print_r($command->params);

// returns all rows of the query result
$rows = $command->queryAll();
```

Indexing Query Results

When you call [[yii\db\Query::all()|all()]], it will return an array of rows which are indexed by consecutive integers. Sometimes you may want to index them differently, such as indexing by a particular column or expression values. You can achieve this goal by calling [[yii\db\Query::indexBy()|indexBy()]] before [[yii\db\Query::all()|all()]]. For example,

```
// returns [100 => ['id' => 100, 'username' => '...', ...], 101 => [...], 103 => [...]
, ...]
$query = (new \yii\db\Query())
    ->from('user')
    ->limit(10)
    ->indexBy('id')
    ->all();
```

To index by expression values, pass an anonymous function to the [[yii\db\Query::indexBy()|indexBy()]] method:

```
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy(function ($row) {
        return $row['id'] . $row['username'];
    })->all();
```

The anonymous function takes a parameter `$row` which contains the current row data and should return a scalar value which will be used as the index value for the current row.

> Note: In contrast to query methods like [[yii\db\Query::groupBy()|groupBy()]] or [[yii\db\Query::orderBy()|orderBy()]] which are converted to SQL and are part of the query, this method works after the data has been fetched from the database. That means that only those column names can be used that have been part of SELECT in your query. Also if you selected a column with table prefix, e.g. `customer.id`, the result set will only contain `id` so you have to call `->indexBy('id')` without table prefix.

Batch Query

When working with large amounts of data, methods such as [[yii\db\Query::all()]] are not suitable because they require loading all data into the memory. To keep the memory requirement low, Yii provides the so-called batch query support. A batch query makes use of the data cursor and fetches data in batches.

Batch query can be used like the following:

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->orderBy('id');

foreach ($query->batch() as $users) {
    // $users is an array of 100 or fewer rows from the user table
}

// or if you want to iterate the row one by one
foreach ($query->each() as $user) {
    // $user represents one row of data from the user table
}
```

The method [[yii\db\Query::batch()]] and [[yii\db\Query::each()]] return an [[yii\db\BatchQueryResult]] object which implements the `Iterator` interface and thus can be used in the `foreach` construct. During the first iteration, a SQL query is made to the database. Data are then fetched in batches in the remaining iterations. By default, the batch size is 100, meaning 100 rows of data are being fetched in each batch. You can change the batch size by passing the first parameter to the `batch()` or `each()` method.

Compared to the [[yii\db\Query::all()]], the batch query only loads 100 rows of data at a time into the memory. If you process the data and then discard it right away, the batch query can help reduce memory usage.

If you specify the query result to be indexed by some column via [[yii\db\Query::indexBy()]], the batch query will still keep the proper index. For example,

```php
$query = (new \yii\db\Query())
    ->from('user')
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users is indexed by the "username" column
}

foreach ($query->each() as $username => $user) {
    // ...
}
```

Active Record

Active Record provides an object-oriented interface for accessing and manipulating data stored in databases. An Active Record class is associated with a database table, an Active Record instance corresponds to a row of that table, and an *attribute* of an Active Record instance represents the value of a particular column in that row. Instead of writing raw SQL statements, you would access Active Record attributes and call Active Record methods to access and manipulate the data stored in database tables.

For example, assume `Customer` is an Active Record class which is associated with the `customer` table and `name` is a column of the `customer` table. You can write the following code to insert a new row into the `customer` table:

```
$customer = new Customer();
$customer->name = 'Qiang';
$customer->save();
```

The above code is equivalent to using the following raw SQL statement for MySQL, which is less intuitive, more error prone, and may even have compatibility problems if you are using a different kind of database:

```
$db->createCommand('INSERT INTO `customer` (`name`) VALUES (:name)', [
    ':name' => 'Qiang',
])->execute();
```

Yii provides the Active Record support for the following relational databases:

- MySQL 4.1 or later: via [[yii\db\ActiveRecord]]
- PostgreSQL 7.3 or later: via [[yii\db\ActiveRecord]]
- SQLite 2 and 3: via [[yii\db\ActiveRecord]]
- Microsoft SQL Server 2008 or later: via [[yii\db\ActiveRecord]]
- Oracle: via [[yii\db\ActiveRecord]]
- CUBRID 9.3 or later: via [[yii\db\ActiveRecord]] (Note that due to a bug in the cubrid PDO extension, quoting of values will not work, so you need CUBRID 9.3 as the client as well as the server)
- Sphinx: via [[yii\sphinx\ActiveRecord]], requires the `yii2-sphinx` extension
- ElasticSearch: via [[yii\elasticsearch\ActiveRecord]], requires the `yii2-elasticsearch` extension

Additionally, Yii also supports using Active Record with the following NoSQL databases:

- Redis 2.6.12 or later: via [[yii\redis\ActiveRecord]], requires the `yii2-redis` extension
- MongoDB 1.3.0 or later: via [[yii\mongodb\ActiveRecord]], requires the `yii2-mongodb` extension

In this tutorial, we will mainly describe the usage of Active Record for relational databases. However, most content described here are also applicable to Active Record for NoSQL databases.

Declaring Active Record Classes

To get started, declare an Active Record class by extending [[yii\db\ActiveRecord]].

Setting a table name

By default each Active Record class is associated with its database table. The [[yii\db\ActiveRecord::tableName()|tableName()]] method returns the table name by converting the class name via [[yii\helpers\Inflector::camel2id()]]. You may override this method if the table is not named after this convention.

Also a default [[yii\db\Connection::$tablePrefix|tablePrefix]] can be applied. For example if [[yii\db\Connection::$tablePrefix|tablePrefix]] is `tbl_`, `Customer` becomes `tbl_customer` and `OrderItem` becomes `tbl_order_item`.

If a table name is given as `{{%TableName}}`, then the percentage character `%` will be replaced with the table prefix. For example, `{{%post}}` becomes `{{tbl_post}}`. The brackets around the table name are used for quoting in an SQL query.

In the following example, we declare an Active Record class named `Customer` for the `customer` database table.

```php
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    const STATUS_INACTIVE = 0;
    const STATUS_ACTIVE = 1;

    /**
     * @return string the name of the table associated with this ActiveRecord class.
     */
    public static function tableName()
    {
        return '{{customer}}';
    }
}
```

Active records are called "models"

Active Record instances are considered as models. For this reason, we usually put Active Record classes under the `app\models` namespace (or other namespaces for keeping model classes).

Because [[yii\db\ActiveRecord]] extends from [[yii\base\Model]], it inherits *all* model features, such as attributes, validation rules, data serialization, etc.

Connecting to Databases

By default, Active Record uses the `db` application component as the [[yii\db\Connection|DB connection]] to access and manipulate the database data. As explained in Database Access Objects, you can configure the `db` component in the application configuration like shown below,

```php
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];
```

If you want to use a different database connection other than the `db` component, you should override the [[yii\db\ActiveRecord::getDb()|getDb()]] method:

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        // use the "db2" application component
        return \Yii::$app->db2;
    }
}
```

Querying Data

After declaring an Active Record class, you can use it to query data from the corresponding database table. The process usually takes the following three steps:

1. Create a new query object by calling the [[yii\db\ActiveRecord::find()]] method;
2. Build the query object by calling query building methods;
3. Call a query method to retrieve data in terms of Active Record instances.

As you can see, this is very similar to the procedure with query builder. The only difference is that instead of using the `new` operator to create a query object, you call [[yii\db\ActiveRecord::find()]] to return a new query object which is of class [[yii\db\ActiveQuery]].

Below are some examples showing how to use Active Query to query data:

```
// return a single customer whose ID is 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::find()
    ->where(['id' => 123])
    ->one();

// return all active customers and order them by their IDs
// SELECT * FROM `customer` WHERE `status` = 1 ORDER BY `id`
$customers = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->orderBy('id')
    ->all();

// return the number of active customers
// SELECT COUNT(*) FROM `customer` WHERE `status` = 1
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// return all customers in an array indexed by customer IDs
// SELECT * FROM `customer`
$customers = Customer::find()
    ->indexBy('id')
    ->all();
```

In the above, `$customer` is a `Customer` object while `$customers` is an array of `Customer` objects. They are all populated with the data retrieved from the `customer` table.

> Info: Because [[yii\db\ActiveQuery]] extends from [[yii\db\Query]], you can use *all* query building methods and query methods as described in the Section Query Builder.

Because it is a common task to query by primary key values or a set of column values, Yii provides two shortcut methods for this purpose:

- [[yii\db\ActiveRecord::findOne()]]: returns a single Active Record instance populated with the first row of the query result.
- [[yii\db\ActiveRecord::findAll()]]: returns an array of Active Record instances populated with *all* query result.

Both methods can take one of the following parameter formats:

- a scalar value: the value is treated as the desired primary key value to be looked for. Yii will determine automatically which column is the primary key column by reading database schema information.
- an array of scalar values: the array is treated as the desired primary key values to be looked for.
- an associative array: the keys are column names and the values are the corresponding

desired column values to be looked for. Please refer to Hash Format for more details.

The following code shows how these methods can be used:

```
// returns a single customer whose ID is 123
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// returns customers whose ID is 100, 101, 123 or 124
// SELECT * FROM `customer` WHERE `id` IN (100, 101, 123, 124)
$customers = Customer::findAll([100, 101, 123, 124]);

// returns an active customer whose ID is 123
// SELECT * FROM `customer` WHERE `id` = 123 AND `status` = 1
$customer = Customer::findOne([
    'id' => 123,
    'status' => Customer::STATUS_ACTIVE,
]);

// returns all inactive customers
// SELECT * FROM `customer` WHERE `status` = 0
$customers = Customer::findAll([
    'status' => Customer::STATUS_INACTIVE,
]);
```

> Note: Neither [[yii\db\ActiveRecord::findOne()]] nor [[yii\db\ActiveQuery::one()]] will add `LIMIT 1` to the generated SQL statement. If your query may return many rows of data, you should call `limit(1)` explicitly to improve the performance, e.g., `Customer::find()->limit(1)->one()`.

Besides using query building methods, you can also write raw SQLs to query data and populate the results into Active Record objects. You can do so by calling the [[yii\db\ActiveRecord::findBySql()]] method:

```
// returns all inactive customers
$sql = 'SELECT * FROM customer WHERE status=:status';
$customers = Customer::findBySql($sql, [':status' => Customer::STATUS_INACTIVE])->all(
);
```

Do not call extra query building methods after calling [[yii\db\ActiveRecord::findBySql()|findBySql()]] as they will be ignored.

Accessing Data

As aforementioned, the data brought back from the database are populated into Active Record instances, and each row of the query result corresponds to a single Active Record instance. You can access the column values by accessing the attributes of the Active Record instances, for example,

```
// "id" and "email" are the names of columns in the "customer" table
$customer = Customer::findOne(123);
$id = $customer->id;
$email = $customer->email;
```

> Note: The Active Record attributes are named after the associated table columns in a case-sensitive manner. Yii automatically defines an attribute in Active Record for every column of the associated table. You should NOT redeclare any of the attributes.

Because Active Record attributes are named after table columns, you may find you are writing PHP code like `$customer->first_name`, which uses underscores to separate words in attribute names if your table columns are named in this way. If you are concerned about code style consistency, you should rename your table columns accordingly (to use camelCase, for example).

Data Transformation

It often happens that the data being entered and/or displayed are in a format which is different from the one used in storing the data in a database. For example, in the database you are storing customers' birthdays as UNIX timestamps (which is not a good design, though), while in most cases you would like to manipulate birthdays as strings in the format of `'YYYY/MM/DD'`. To achieve this goal, you can define *data transformation* methods in the `Customer` Active Record class like the following:

```
class Customer extends ActiveRecord
{
    // ...

    public function getBirthdayText()
    {
        return date('Y/m/d', $this->birthday);
    }

    public function setBirthdayText($value)
    {
        $this->birthday = strtotime($value);
    }
}
```

Now in your PHP code, instead of accessing `$customer->birthday`, you would access `$customer->birthdayText`, which will allow you to input and display customer birthdays in the format of `'YYYY/MM/DD'`.

> Tip: The above example shows a generic way of transforming data in different formats. If you are working with date values, you may use DateValidator and [[yii\jui\DatePicker|DatePicker]], which is easier to use and more powerful.

Retrieving Data in Arrays

While retrieving data in terms of Active Record objects is convenient and flexible, it is not always desirable when you have to bring back a large amount of data due to the big memory footprint. In this case, you can retrieve data using PHP arrays by calling [[yii\db\ActiveQuery::asArray()|asArray()]] before executing a query method:

```
// return all customers
// each customer is returned as an associative array
$customers = Customer::find()
    ->asArray()
    ->all();
```

> Note: While this method saves memory and improves performance, it is closer to the lower DB abstraction layer and you will lose most of the Active Record features. A very important distinction lies in the data type of the column values. When you return data in Active Record instances, column values will be automatically typecast according to the actual column types; on the other hand when you return data in arrays, column values will be strings (since they are the result of PDO without any processing), regardless their actual column types.

Retrieving Data in Batches

In Query Builder, we have explained that you may use *batch query* to minimize your memory usage when querying a large amount of data from the database. You may use the same technique in Active Record. For example,

```
// fetch 10 customers at a time
foreach (Customer::find()->batch(10) as $customers) {
    // $customers is an array of 10 or fewer Customer objects
}

// fetch 10 customers at a time and iterate them one by one
foreach (Customer::find()->each(10) as $customer) {
    // $customer is a Customer object
}

// batch query with eager loading
foreach (Customer::find()->with('orders')->each() as $customer) {
    // $customer is a Customer object with the 'orders' relation populated
}
```

Saving Data

Using Active Record, you can easily save data to the database by taking the following steps:

1. Prepare an Active Record instance
2. Assign new values to Active Record attributes
3. Call [[yii\db\ActiveRecord::save()]] to save the data into database.

For example,

```
// insert a new row of data
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();

// update an existing row of data
$customer = Customer::findOne(123);
$customer->email = 'james@newexample.com';
$customer->save();
```

The [[yii\db\ActiveRecord::save()|save()]] method can either insert or update a row of data, depending on the state of the Active Record instance. If the instance is newly created via the `new` operator, calling [[yii\db\ActiveRecord::save()|save()]] will cause insertion of a new row; If the instance is the result of a query method, calling [[yii\db\ActiveRecord::save()|save()]] will update the row associated with the instance.

You can differentiate the two states of an Active Record instance by checking its [[yii\db\ActiveRecord::isNewRecord|isNewRecord]] property value. This property is also used by [[yii\db\ActiveRecord::save()|save()]] internally as follows:

```
public function save($runValidation = true, $attributeNames = null)
{
    if ($this->getIsNewRecord()) {
        return $this->insert($runValidation, $attributeNames);
    } else {
        return $this->update($runValidation, $attributeNames) !== false;
    }
}
```

> Tip: You can call [[yii\db\ActiveRecord::insert()|insert()]] or [[yii\db\ActiveRecord::update()|update()]] directly to insert or update a row.

Data Validation

Because [[yii\db\ActiveRecord]] extends from [[yii\base\Model]], it shares the same data validation feature. You can declare validation rules by overriding the [[yii\db\ActiveRecord::rules()|rules()]] method and perform data validation by calling the [[yii\db\ActiveRecord::validate()|validate()]] method.

When you call [[yii\db\ActiveRecord::save()|save()]], by default it will call [[yii\db\ActiveRecord::validate()|validate()]] automatically. Only when the validation passes, will it actually save the data; otherwise it will simply return `false`, and you can check the [[yii\db\ActiveRecord::errors|errors]] property to retrieve the validation error messages.

> Tip: If you are certain that your data do not need validation (e.g., the data comes from trustable sources), you can call `save(false)` to skip the validation.

Massive Assignment

Like normal models, Active Record instances also enjoy the massive assignment feature. Using this feature, you can assign values to multiple attributes of an Active Record instance in a single PHP statement, like shown below. Do remember that only safe attributes can be massively assigned, though.

```
$values = [
    'name' => 'James',
    'email' => 'james@example.com',
];

$customer = new Customer();

$customer->attributes = $values;
$customer->save();
```

Updating Counters

It is a common task to increment or decrement a column in a database table. We call these columns "counter columns". You can use [[yii\db\ActiveRecord::updateCounters()|updateCounters()]] to update one or multiple counter columns. For example,

```
$post = Post::findOne(100);

// UPDATE `post` SET `view_count` = `view_count` + 1 WHERE `id` = 100
$post->updateCounters(['view_count' => 1]);
```

> Note: If you use [[yii\db\ActiveRecord::save()]] to update a counter column, you may end up with inaccurate result, because it is likely the same counter is being saved by multiple requests which read and write the same counter value.

Dirty Attributes

When you call [[yii\db\ActiveRecord::save()|save()]] to save an Active Record instance, only *dirty attributes* are being saved. An attribute is considered *dirty* if its value has been modified since it was loaded from DB or saved to DB most recently. Note that data validation will be performed regardless if the Active Record instance has dirty attributes or not.

Active Record automatically maintains the list of dirty attributes. It does so by maintaining an older version of the attribute values and comparing them with the latest one. You can call [[yii\db\ActiveRecord::getDirtyAttributes()]] to get the attributes that are currently dirty. You can also call [[yii\db\ActiveRecord::markAttributeDirty()]] to explicitly mark an attribute as dirty.

If you are interested in the attribute values prior to their most recent modification, you may call [[yii\db\ActiveRecord::getOldAttributes()|getOldAttributes()]] or [[yii\db\ActiveRecord::getOldAttribute()|getOldAttribute()]].

> Note: The comparison of old and new values will be done using the `===` operator so a value will be considered dirty even if it has the same value but a different type. This is often the case when the model receives user input from HTML forms where every value is represented as a string. To ensure the correct type for e.g. integer values you may apply a validation filter: `['attributeName', 'filter', 'filter' => 'intval']`. This works with all the typecasting functions of PHP like intval(), floatval(), boolval, etc...

Default Attribute Values

Some of your table columns may have default values defined in the database. Sometimes, you may want to pre-populate your Web form for an Active Record instance with these default values. To avoid writing the same default values again, you can call [[yii\db\ActiveRecord::loadDefaultValues()|loadDefaultValues()]] to populate the DB-defined default values into the corresponding Active Record attributes:

```
$customer = new Customer();
$customer->loadDefaultValues();
// $customer->xyz will be assigned the default value declared when defining the "xyz"
column
```

Attributes Typecasting

Being populated by query results [[yii\db\ActiveRecord]] performs automatic typecast for its attribute values, using information from database table schema. This allows data retrieved from table column declared as integer to be populated in ActiveRecord instance with PHP integer, boolean with boolean and so on. However, typecasting mechanism has several limitations:

- Float values are not be converted and will be represented as strings, otherwise they may loose precision.
- Conversion of the integer values depends on the integer capacity of the operation system you use. In particular: values of column declared as 'unsigned integer' or 'big integer' will be converted to PHP integer only at 64-bit operation system, while on 32-bit ones - they will be represented as strings.

Note that attribute typecast is performed only during populating ActiveRecord instance from query result. There is no automatic conversion for the values loaded from HTTP request or set directly via property access. The table schema will also be used while preparing SQL statements for the ActiveRecord data saving, ensuring values are bound to the query with correct type. However, ActiveRecord instance attribute values will not be converted during saving process.

> Tip: you may use [[yii\behaviors\AttributeTypecastBehavior]] to facilitate attribute values typecasting on ActiveRecord validation or saving.

Updating Multiple Rows

The methods described above all work on individual Active Record instances, causing inserting or updating of individual table rows. To update multiple rows simultaneously, you should call [[yii\db\ActiveRecord::updateAll()|updateAll()]], instead, which is a static method.

```
// UPDATE `customer` SET `status` = 1 WHERE `email` LIKE `%@example.com%`
Customer::updateAll(['status' => Customer::STATUS_ACTIVE], ['like', 'email', '@example
.com']);
```

Similarly, you can call [[yii\db\ActiveRecord::updateAllCounters()|updateAllCounters()]] to update counter columns of multiple rows at the same time.

```
// UPDATE `customer` SET `age` = `age` + 1
Customer::updateAllCounters(['age' => 1]);
```

Deleting Data

To delete a single row of data, first retrieve the Active Record instance corresponding to that row and then call the [[yii\db\ActiveRecord::delete()]] method.

```
$customer = Customer::findOne(123);
$customer->delete();
```

You can call [[yii\db\ActiveRecord::deleteAll()]] to delete multiple or all rows of data. For example,

```
Customer::deleteAll(['status' => Customer::STATUS_INACTIVE]);
```

> Note: Be very careful when calling [[yii\db\ActiveRecord::deleteAll()|deleteAll()]] because it may totally erase all data from your table if you make a mistake in specifying the condition.

Active Record Life Cycles

It is important to understand the life cycles of Active Record when it is used for different purposes. During each life cycle, a certain sequence of methods will be invoked, and you can override these methods to get a chance to customize the life cycle. You can also respond to certain Active Record events triggered during a life cycle to inject your custom code. These events are especially useful when you are developing Active Record behaviors which need to customize Active Record life cycles.

In the following, we will summarize the various Active Record life cycles and the methods/events that are involved in the life cycles.

New Instance Life Cycle

When creating a new Active Record instance via the `new` operator, the following life cycle will happen:

1. Class constructor.
2. [[yii\db\ActiveRecord::init()|init()]]: triggers an [[yii\db\ActiveRecord::EVENT_INIT|EVENT_INIT]] event.

Querying Data Life Cycle

When querying data through one of the querying methods, each newly populated Active Record will undergo the following life cycle:

1. Class constructor.
2. [[yii\db\ActiveRecord::init()|init()]]: triggers an [[yii\db\ActiveRecord::EVENT_INIT|EVENT_INIT]] event.
3. [[yii\db\ActiveRecord::afterFind()|afterFind()]]: triggers an [[yii\db\ActiveRecord::EVENT_AFTER_FIND|EVENT_AFTER_FIND]] event.

Saving Data Life Cycle

When calling [[yii\db\ActiveRecord::save()|save()]] to insert or update an Active Record instance, the following life cycle will happen:

1. [[yii\db\ActiveRecord::beforeValidate()|beforeValidate()]]: triggers an [[yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE|EVENT_BEFORE_VALIDATE]] event. If the method returns `false` or [[yii\base\ModelEvent::isValid]] is `false`, the rest of the steps will be skipped.
2. Performs data validation. If data validation fails, the steps after Step 3 will be skipped.
3. [[yii\db\ActiveRecord::afterValidate()|afterValidate()]]: triggers an [[yii\db\ActiveRecord::EVENT_AFTER_VALIDATE|EVENT_AFTER_VALIDATE]] event.
4. [[yii\db\ActiveRecord::beforeSave()|beforeSave()]]: triggers an [[yii\db\ActiveRecord::EVENT_BEFORE_INSERT|EVENT_BEFORE_INSERT]] or [[yii\db\ActiveRecord::EVENT_BEFORE_UPDATE|EVENT_BEFORE_UPDATE]] event. If the method returns `false` or [[yii\base\ModelEvent::isValid]] is `false`, the rest of the steps will be skipped.
5. Performs the actual data insertion or updating.
6. [[yii\db\ActiveRecord::afterSave()|afterSave()]]: triggers an [[yii\db\ActiveRecord::EVENT_AFTER_INSERT|EVENT_AFTER_INSERT]] or [[yii\db\ActiveRecord::EVENT_AFTER_UPDATE|EVENT_AFTER_UPDATE]] event.

Deleting Data Life Cycle

When calling [[yii\db\ActiveRecord::delete()|delete()]] to delete an Active Record instance, the following life cycle will happen:

1. [[yii\db\ActiveRecord::beforeDelete()|beforeDelete()]]: triggers an [[yii\db\ActiveRecord::EVENT_BEFORE_DELETE|EVENT_BEFORE_DELETE]] event. If the method returns `false` or [[yii\base\ModelEvent::isValid]] is `false`, the rest of the steps will be skipped.
2. Performs the actual data deletion.
3. [[yii\db\ActiveRecord::afterDelete()|afterDelete()]]: triggers an [[yii\db\ActiveRecord::EVENT_AFTER_DELETE|EVENT_AFTER_DELETE]] event.

> Note: Calling any of the following methods will NOT initiate any of the above life cycles because they work on the database directly and not on a record basis:
>
> - [[yii\db\ActiveRecord::updateAll()]]
> - [[yii\db\ActiveRecord::deleteAll()]]
> - [[yii\db\ActiveRecord::updateCounters()]]
> - [[yii\db\ActiveRecord::updateAllCounters()]]

Refreshing Data Life Cycle

When calling [[yii\db\ActiveRecord::refresh()|refresh()]] to refresh an Active Record instance, the [[yii\db\ActiveRecord::EVENT_AFTER_REFRESH|EVENT_AFTER_REFRESH]] event is triggered if refresh is successful and the method returns `true`.

Working with Transactions

There are two ways of using transactions while working with Active Record.

The first way is to explicitly enclose Active Record method calls in a transactional block, like shown below,

```
$customer = Customer::findOne(123);

Customer::getDb()->transaction(function($db) use ($customer) {
    $customer->id = 200;
    $customer->save();
    // ...other DB operations...
});

// or alternatively

$transaction = Customer::getDb()->beginTransaction();
try {
    $customer->id = 200;
    $customer->save();
    // ...other DB operations...
    $transaction->commit();
} catch(\Exception $e) {
    $transaction->rollBack();
    throw $e;
} catch(\Throwable $e) {
    $transaction->rollBack();
    throw $e;
}
```

> Note: in the above code we have two catch-blocks for compatibility with PHP 5.x and PHP 7.x. `\Exception` implements the `\Throwable` interface since PHP 7.0, so you can skip the part with `\Exception` if your app uses only PHP 7.0 and higher.

The second way is to list the DB operations that require transactional support in the [[yii\db\ActiveRecord::transactions()]] method. For example,

```
class Customer extends ActiveRecord
{
    public function transactions()
    {
        return [
            'admin' => self::OP_INSERT,
            'api' => self::OP_INSERT | self::OP_UPDATE | self::OP_DELETE,
            // the above is equivalent to the following:
            // 'api' => self::OP_ALL,
        ];
    }
}
```

The [[yii\db\ActiveRecord::transactions()]] method should return an array whose keys are scenario names and values are the corresponding operations that should be enclosed within transactions. You should use the following constants to refer to different DB operations:

- [[yii\db\ActiveRecord::OP_INSERT|OP_INSERT]]: insertion operation performed by [[yii\db\ActiveRecord::insert()|insert()]];
- [[yii\db\ActiveRecord::OP_UPDATE|OP_UPDATE]]: update operation performed by [[yii\db\ActiveRecord::update()|update()]];
- [[yii\db\ActiveRecord::OP_DELETE|OP_DELETE]]: deletion operation performed by [[yii\db\ActiveRecord::delete()|delete()]].

Use the `|` operators to concatenate the above constants to indicate multiple operations. You may also use the shortcut constant [[yii\db\ActiveRecord::OP_ALL|OP_ALL]] to refer to all three operations above.

Transactions that are created using this method will be started before calling [[yii\db\ActiveRecord::beforeSave()|beforeSave()]] and will be committed after [[yii\db\ActiveRecord::afterSave()|afterSave()]] has run.

Optimistic Locks

Optimistic locking is a way to prevent conflicts that may occur when a single row of data is being updated by multiple users. For example, both user A and user B are editing the same wiki article at the same time. After user A saves his edits, user B clicks on the "Save" button in an attempt to save his edits as well. Because user B was actually working on an outdated version of the article, it would be desirable to have a way to prevent him from saving the article and show him some hint message.

Optimistic locking solves the above problem by using a column to record the version number of each row. When a row is being saved with an outdated version number, a [[yii\db\StaleObjectException]] exception will be thrown, which prevents the row from being saved. Optimistic locking is only supported when you update or delete an existing row of data using [[yii\db\ActiveRecord::update()]] or [[yii\db\ActiveRecord::delete()]], respectively.

To use optimistic locking,

1. Create a column in the DB table associated with the Active Record class to store the version number of each row. The column should be of big integer type (in MySQL it would be `BIGINT DEFAULT 0`).
2. Override the [[yii\db\ActiveRecord::optimisticLock()]] method to return the name of this column.
3. In the Web form that takes user inputs, add a hidden field to store the current version number of the row being updated. Be sure your version attribute has input validation rules and validates successfully.
4. In the controller action that updates the row using Active Record, try and catch the [[yii\db\StaleObjectException]] exception. Implement necessary business logic (e.g.

merging the changes, prompting staled data) to resolve the conflict.

For example, assume the version column is named as `version`. You can implement optimistic locking with the code like the following.

```
// ------ view code -------

use yii\helpers\Html;

// ...other input fields
echo Html::activeHiddenInput($model, 'version');


// ------ controller code -------

use yii\db\StaleObjectException;

public function actionUpdate($id)
{
    $model = $this->findModel($id);

    try {
        if ($model->load(Yii::$app->request->post()) && $model->save()) {
            return $this->redirect(['view', 'id' => $model->id]);
        } else {
            return $this->render('update', [
                'model' => $model,
            ]);
        }
    } catch (StaleObjectException $e) {
        // logic to resolve the conflict
    }
}
```

Working with Relational Data

Besides working with individual database tables, Active Record is also capable of bringing together related data, making them readily accessible through the primary data. For example, the customer data is related with the order data because one customer may have placed one or multiple orders. With appropriate declaration of this relation, you'll be able to access a customer's order information using the expression `$customer->orders` which gives back the customer's order information in terms of an array of `Order` Active Record instances.

Declaring Relations

To work with relational data using Active Record, you first need to declare relations in Active Record classes. The task is as simple as declaring a *relation method* for every interested relation, like the following,

```
class Customer extends ActiveRecord
{
    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    // ...

    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id']);
    }
}
```

In the above code, we have declared an `orders` relation for the `Customer` class, and a `customer` relation for the `Order` class.

Each relation method must be named as `getXyz`. We call `xyz` (the first letter is in lower case) the *relation name*. Note that relation names are *case sensitive*.

While declaring a relation, you should specify the following information:

- the multiplicity of the relation: specified by calling either [[yii\db\ActiveRecord::hasMany()|hasMany()]] or [[yii\db\ActiveRecord::hasOne()|hasOne()]]. In the above example you may easily read in the relation declarations that a customer has many orders while an order only has one customer.
- the name of the related Active Record class: specified as the first parameter to either [[yii\db\ActiveRecord::hasMany()|hasMany()]] or [[yii\db\ActiveRecord::hasOne()|hasOne()]]. A recommended practice is to call `Xyz::className()` to get the class name string so that you can receive IDE auto-completion support as well as error detection at compiling stage.
- the link between the two types of data: specifies the column(s) through which the two types of data are related. The array values are the columns of the primary data (represented by the Active Record class that you are declaring relations), while the array keys are the columns of the related data.

> An easy rule to remember this is, as you see in the example above, you write the column that belongs to the related Active Record directly next to it. You see there that `customer_id` is a property of `Order` and `id` is a property of `Customer`.

Accessing Relational Data

After declaring relations, you can access relational data through relation names. This is just like accessing an object property defined by the relation method. For this reason, we call it *relation property*. For example,

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
// $orders is an array of Order objects
$orders = $customer->orders;
```

> Info: When you declare a relation named `xyz` via a getter method `getXyz()`, you will be able to access `xyz` like an object property. Note that the name is case sensitive.

If a relation is declared with [[yii\db\ActiveRecord::hasMany()|hasMany()]], accessing this relation property will return an array of the related Active Record instances; if a relation is declared with [[yii\db\ActiveRecord::hasOne()|hasOne()]], accessing the relation property will return the related Active Record instance or `null` if no related data is found.

When you access a relation property for the first time, a SQL statement will be executed, like shown in the above example. If the same property is accessed again, the previous result will be returned without re-executing the SQL statement. To force re-executing the SQL statement, you should unset the relation property first: `unset($customer->orders)`.

> Note: While this concept looks similar to the object property feature, there is an important difference. For normal object properties the property value is of the same type as the defining getter method. A relation method however returns an [[yii\db\ActiveQuery]] instance, while accessing a relation property will either return a [[yii\db\ActiveRecord]] instance or an array of these.
>
> ```
> $customer->orders; // is an array of `Order` objects
> $customer->getOrders(); // returns an ActiveQuery instance
> ```
>
> This is useful for creating customized queries, which is described in the next section.

Dynamic Relational Query

Because a relation method returns an instance of [[yii\db\ActiveQuery]], you can further build this query using query building methods before performing DB query. For example,

```
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 200 ORDER BY `id`
$orders = $customer->getOrders()
    ->where(['>', 'subtotal', 200])
    ->orderBy('id')
    ->all();
```

Unlike accessing a relation property, each time you perform a dynamic relational query via a relation method, a SQL statement will be executed, even if the same dynamic relational query was performed before.

Sometimes you may even want to parametrize a relation declaration so that you can more easily perform dynamic relational query. For example, you may declare a `bigOrders` relation as follows,

```
class Customer extends ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

Then you will be able to perform the following relational queries:

```
// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 200 ORDER BY `id`
$orders = $customer->getBigOrders(200)->all();

// SELECT * FROM `order` WHERE `customer_id` = 123 AND `subtotal` > 100 ORDER BY `id`
$orders = $customer->bigOrders;
```

Relations via a Junction Table

In database modelling, when the multiplicity between two related tables is many-to-many, a junction table is usually introduced. For example, the `order` table and the `item` table may be related via a junction table named `order_item`. One order will then correspond to multiple order items, while one product item will also correspond to multiple order items.

When declaring such relations, you would call either [[yii\db\ActiveQuery::via()|via()]] or [[yii\db\ActiveQuery::viaTable()|viaTable()]] to specify the junction table. The difference between [[yii\db\ActiveQuery::via()|via()]] and [[yii\db\ActiveQuery::viaTable()|viaTable()]] is that the former specifies the junction table in terms of an existing relation name while the latter directly uses the junction table. For example,

```
class Order extends ActiveRecord
{
    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

or alternatively,

```
class Order extends ActiveRecord
{
    public function getOrderItems()
    {
        return $this->hasMany(OrderItem::className(), ['order_id' => 'id']);
    }

    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->via('orderItems');
    }
}
```

The usage of relations declared with a junction table is the same as that of normal relations. For example,

```
// SELECT * FROM `order` WHERE `id` = 100
$order = Order::findOne(100);

// SELECT * FROM `order_item` WHERE `order_id` = 100
// SELECT * FROM `item` WHERE `item_id` IN (...)
// returns an array of Item objects
$items = $order->items;
```

Lazy Loading and Eager Loading

In Accessing Relational Data, we explained that you can access a relation property of an Active Record instance like accessing a normal object property. A SQL statement will be executed only when you access the relation property the first time. We call such relational data accessing method *lazy loading*. For example,

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$orders = $customer->orders;

// no SQL executed
$orders2 = $customer->orders;
```

Lazy loading is very convenient to use. However, it may suffer from a performance issue when you need to access the same relation property of multiple Active Record instances. Consider the following code example. How many SQL statements will be executed?

```
// SELECT * FROM `customer` LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SELECT * FROM `order` WHERE `customer_id` = ...
    $orders = $customer->orders;
}
```

As you can see from the code comment above, there are 101 SQL statements being executed! This is because each time you access the `orders` relation property of a different `Customer` object in the for-loop, a SQL statement will be executed.

To solve this performance problem, you can use the so-called *eager loading* approach as shown below,

```
// SELECT * FROM `customer` LIMIT 100;
// SELECT * FROM `orders` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->with('orders')
    ->limit(100)
    ->all();

foreach ($customers as $customer) {
    // no SQL executed
    $orders = $customer->orders;
}
```

By calling [[yii\db\ActiveQuery::with()]], you instruct Active Record to bring back the orders for the first 100 customers in one single SQL statement. As a result, you reduce the number of the executed SQL statements from 101 to 2!

You can eagerly load one or multiple relations. You can even eagerly load *nested relations*. A nested relation is a relation that is declared within a related Active Record class. For example, `Customer` is related with `Order` through the `orders` relation, and `Order` is related with `Item` through the `items` relation. When querying for `Customer`, you can eagerly load `items` using the nested relation notation `orders.items`.

The following code shows different usage of [[yii\db\ActiveQuery::with()|with()]]. We assume the `Customer` class has two relations `orders` and `country`, while the `Order` class has one relation `items`.

```
// eager loading both "orders" and "country"
$customers = Customer::find()->with('orders', 'country')->all();
// equivalent to the array syntax below
$customers = Customer::find()->with(['orders', 'country'])->all();
// no SQL executed
$orders= $customers[0]->orders;
// no SQL executed
$country = $customers[0]->country;

// eager loading "orders" and the nested relation "orders.items"
$customers = Customer::find()->with('orders.items')->all();
// access the items of the first order of the first customer
// no SQL executed
$items = $customers[0]->orders[0]->items;
```

You can eagerly load deeply nested relations, such as `a.b.c.d`. All parent relations will be eagerly loaded. That is, when you call [[yii\db\ActiveQuery::with()|with()]] using `a.b.c.d`, you will eagerly load `a`, `a.b`, `a.b.c` and `a.b.c.d`.

> Info: In general, when eagerly loading `N` relations among which `M` relations are defined with a junction table, a total number of `N+M+1` SQL statements will be executed. Note that a nested relation `a.b.c.d` counts as 4 relations.

When eagerly loading a relation, you can customize the corresponding relational query using an anonymous function. For example,

```
// find customers and bring back together their country and active orders
// SELECT * FROM `customer`
// SELECT * FROM `country` WHERE `id` IN (...)
// SELECT * FROM `order` WHERE `customer_id` IN (...) AND `status` = 1
$customers = Customer::find()->with([
    'country',
    'orders' => function ($query) {
        $query->andWhere(['status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

When customizing the relational query for a relation, you should specify the relation name as an array key and use an anonymous function as the corresponding array value. The anonymous function will receive a `$query` parameter which represents the [[yii\db\ActiveQuery]] object used to perform the relational query for the relation. In the code example above, we are modifying the relational query by appending an additional condition about order status.

> Note: If you call [[yii\db\Query::select()|select()]] while eagerly loading relations, you have to make sure the columns referenced in the relation declarations are being selected. Otherwise, the related models may not be loaded properly. For example,
>
> ```
> $orders = Order::find()->select(['id', 'amount'])->with('customer')->all();
> // $orders[0]->customer is always `null`. To fix the problem, you should do the f
> ollowing:
> $orders = Order::find()->select(['id', 'amount', 'customer_id'])->with('customer'
> )->all();
> ```

Joining with Relations

> Note: The content described in this subsection is only applicable to relational databases, such as MySQL, PostgreSQL, etc.

The relational queries that we have described so far only reference the primary table columns when querying for the primary data. In reality we often need to reference columns in the related tables. For example, we may want to bring back the customers who have at least one active order. To solve this problem, we can build a join query like the following:

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id`
// WHERE `order`.`status` = 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()
    ->select('customer.*')
    ->leftJoin('order', ``order`.`customer_id` = `customer`.`id``)
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->with('orders')
    ->all();
```

> Note: It is important to disambiguate column names when building relational queries involving JOIN SQL statements. A common practice is to prefix column names with their corresponding table names.

However, a better approach is to exploit the existing relation declarations by calling [[yii\db\ActiveQuery::joinWith()]]:

```
$customers = Customer::find()
    ->joinWith('orders')
    ->where(['order.status' => Order::STATUS_ACTIVE])
    ->all();
```

Both approaches execute the same set of SQL statements. The latter approach is much cleaner and drier, though.

By default, [[yii\db\ActiveQuery::joinWith()|joinWith()]] will use `LEFT JOIN` to join the primary table with the related table. You can specify a different join type (e.g. `RIGHT JOIN`) via its third parameter `$joinType`. If the join type you want is `INNER JOIN`, you can simply call [[yii\db\ActiveQuery::innerJoinWith()|innerJoinWith()]], instead.

Calling [[yii\db\ActiveQuery::joinWith()|joinWith()]] will eagerly load the related data by default. If you do not want to bring in the related data, you can specify its second parameter `$eagerLoading` as `false`.

Like [[yii\db\ActiveQuery::with()|with()]], you can join with one or multiple relations; you may customize the relation queries on-the-fly; you may join with nested relations; and you may mix the use of [[yii\db\ActiveQuery::with()|with()]] and [[yii\db\ActiveQuery::joinWith()|joinWith()]]. For example,

```
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->andWhere(['>', 'subtotal', 100]);
    },
])->with('country')
    ->all();
```

Sometimes when joining two tables, you may need to specify some extra conditions in the `ON` part of the JOIN query. This can be done by calling the [[yii\db\ActiveQuery::onCondition()]] method like the following:

```
// SELECT `customer`.* FROM `customer`
// LEFT JOIN `order` ON `order`.`customer_id` = `customer`.`id` AND `order`.`status` =
 1
//
// SELECT * FROM `order` WHERE `customer_id` IN (...)
$customers = Customer::find()->joinWith([
    'orders' => function ($query) {
        $query->onCondition(['order.status' => Order::STATUS_ACTIVE]);
    },
])->all();
```

This above query brings back *all* customers, and for each customer it brings back all active orders. Note that this differs from our earlier example which only brings back customers who have at least one active order.

> Info: When [[yii\db\ActiveQuery]] is specified with a condition via [[yii\db\ActiveQuery::onCondition()|onCondition()]], the condition will be put in the `ON` part if the query involves a JOIN query. If the query does not involve JOIN, the on-condition will be automatically appended to the `WHERE` part of the query. Thus it may only contain conditions including columns of the related table.

Relation table aliases

As noted before, when using JOIN in a query, we need to disambiguate column names. Therefor often an alias is defined for a table. Setting an alias for the relational query would be possible by customizing the relation query in the following way:

```
$query->joinWith([
    'orders' => function ($q) {
        $q->from(['o' => Order::tableName()]);
    },
])
```

This however looks very complicated and involves either hardcoding the related objects table name or calling `Order::tableName()`. Since version 2.0.7, Yii provides a shortcut for this. You may now define and use the alias for the relation table like the following:

```
// join the orders relation and sort the result by orders.id
$query->joinWith(['orders o'])->orderBy('o.id');
```

The above syntax works for simple relations. If you need an alias for an intermediate table when joining over nested relations, e.g. `$query->joinWith(['orders.product'])`, you need to nest the joinWith calls like in the following example:

```
$query->joinWith(['orders o' => function($q) {
        $q->joinWith('product p');
    }])
    ->where('o.amount > 100');
```

Inverse Relations

Relation declarations are often reciprocal between two Active Record classes. For example, `Customer` is related to `Order` via the `orders` relation, and `Order` is related back to `Customer` via the `customer` relation.

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}

class Order extends ActiveRecord
{
    public function getCustomer()
    {
        return $this->hasOne(Customer::className(), ['id' => 'customer_id']);
    }
}
```

Now consider the following piece of code:

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// SELECT * FROM `customer` WHERE `id` = 123
$customer2 = $order->customer;

// displays "not the same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

We would think `$customer` and `$customer2` are the same, but they are not! Actually they do contain the same customer data, but they are different objects. When accessing `$order->customer`, an extra SQL statement is executed to populate a new object `$customer2`.

To avoid the redundant execution of the last SQL statement in the above example, we should tell Yii that `customer` is an *inverse relation* of `orders` by calling the [[yii\db\ActiveQuery::inverseOf()|inverseOf()]] method like shown below:

```
class Customer extends ActiveRecord
{
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id'])->inverseOf(
'customer');
    }
}
```

With this modified relation declaration, we will have:

```
// SELECT * FROM `customer` WHERE `id` = 123
$customer = Customer::findOne(123);

// SELECT * FROM `order` WHERE `customer_id` = 123
$order = $customer->orders[0];

// No SQL will be executed
$customer2 = $order->customer;

// displays "same"
echo $customer2 === $customer ? 'same' : 'not the same';
```

> Note: Inverse relations cannot be defined for relations involving a junction table. That is, if a relation is defined with [[yii\db\ActiveQuery::via()|via()]] or [[yii\db\ActiveQuery::viaTable()|viaTable()]], you should not call [[yii\db\ActiveQuery::inverseOf()|inverseOf()]] further.

Saving Relations

When working with relational data, you often need to establish relationships between different data or destroy existing relationships. This requires setting proper values for the columns that define the relations. Using Active Record, you may end up writing the code like the following:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

// setting the attribute that defines the "customer" relation in Order
$order->customer_id = $customer->id;
$order->save();
```

Active Record provides the [[yii\db\ActiveRecord::link()|link()]] method that allows you to accomplish this task more nicely:

```
$customer = Customer::findOne(123);
$order = new Order();
$order->subtotal = 100;
// ...

$order->link('customer', $customer);
```

The [[yii\db\ActiveRecord::link()|link()]] method requires you to specify the relation name and the target Active Record instance that the relationship should be established with. The method will modify the values of the attributes that link two Active Record instances and save them to the database. In the above example, it will set the `customer_id` attribute of the `Order` instance to be the value of the `id` attribute of the `Customer` instance and then save it to the database.

> Note: You cannot link two newly created Active Record instances.

The benefit of using [[yii\db\ActiveRecord::link()|link()]] is even more obvious when a relation is defined via a junction table. For example, you may use the following code to link an `Order` instance with an `Item` instance:

```
$order->link('items', $item);
```

The above code will automatically insert a row in the `order_item` junction table to relate the order with the item.

> Info: The [[yii\db\ActiveRecord::link()|link()]] method will NOT perform any data validation while saving the affected Active Record instance. It is your responsibility to validate any input data before calling this method.

The opposite operation to [[yii\db\ActiveRecord::link()|link()]] is [[yii\db\ActiveRecord::unlink()|unlink()]] which breaks an existing relationship between two Active Record instances. For example,

```
$customer = Customer::find()->with('orders')->where(['id' => 123])->one();
$customer->unlink('orders', $customer->orders[0]);
```

By default, the [[yii\db\ActiveRecord::unlink()|unlink()]] method will set the foreign key value(s) that specify the existing relationship to be `null`. You may, however, choose to delete the table row that contains the foreign key value by passing the `$delete` parameter as `true` to the method.

When a junction table is involved in a relation, calling [[yii\db\ActiveRecord::unlink()|unlink()]] will cause the foreign keys in the junction table to be cleared, or the deletion of the corresponding row in the junction table if `$delete` is `true`.

Cross-Database Relations

Active Record allows you to declare relations between Active Record classes that are powered by different databases. The databases can be of different types (e.g. MySQL and PostgreSQL, or MS SQL and MongoDB), and they can run on different servers. You can use the same syntax to perform relational queries. For example,

```
// Customer is associated with the "customer" table in a relational database (e.g. MyS
QL)
class Customer extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
    }

    public function getComments()
    {
        // a customer has many comments
        return $this->hasMany(Comment::className(), ['customer_id' => 'id']);
    }
}

// Comment is associated with the "comment" collection in a MongoDB database
class Comment extends \yii\mongodb\ActiveRecord
{
    public static function collectionName()
    {
        return 'comment';
    }

    public function getCustomer()
    {
        // a comment has one customer
        return $this->hasOne(Customer::className(), ['id' => 'customer_id']);
    }
}

$customers = Customer::find()->with('comments')->all();
```

You can use most of the relational query features that have been described in this section.

> Note: Usage of [[yii\db\ActiveQuery::joinWith()|joinWith()]] is limited to databases that allow cross-database JOIN queries. For this reason, you cannot use this method in the above example because MongoDB does not support JOIN.

Customizing Query Classes

By default, all Active Record queries are supported by [[yii\db\ActiveQuery]]. To use a customized query class in an Active Record class, you should override the [[yii\db\ActiveRecord::find()]] method and return an instance of your customized query class. For example,

```php
// file Comment.php
namespace app\models;

use yii\db\ActiveRecord;

class Comment extends ActiveRecord
{
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}
```

Now whenever you are performing a query (e.g. `find()`, `findOne()`) or defining a relation (e.g. `hasOne()`) with `Comment`, you will be calling an instance of `CommentQuery` instead of `ActiveQuery`.

You now have to define the `CommentQuery` class, which can be customized in many creative ways to improve your query building experience. For example,

```php
// file CommentQuery.php
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    // conditions appended by default (can be skipped)
    public function init()
    {
        $this->andOnCondition(['deleted' => false]);
        parent::init();
    }

    // ... add customized query methods here ...

    public function active($state = true)
    {
        return $this->andOnCondition(['active' => $state]);
    }
}
```

> Note: Instead of calling [[yii\db\ActiveQuery::onCondition()|onCondition()]], you usually should call [[yii\db\ActiveQuery::andOnCondition()|andOnCondition()]] or [[yii\db\ActiveQuery::orOnCondition()|orOnCondition()]] to append additional conditions when defining new query building methods so that any existing conditions are not overwritten.

This allows you to write query building code like the following:

```
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

> Tip: In big projects, it is recommended that you use customized query classes to hold most query-related code so that the Active Record classes can be kept clean.

You can also use the new query building methods when defining relations about `Comment` or performing relational query:

```
class Customer extends \yii\db\ActiveRecord
{
    public function getActiveComments()
    {
        return $this->hasMany(Comment::className(), ['customer_id' => 'id'])->active()
;
    }
}

$customers = Customer::find()->joinWith('activeComments')->all();

// or alternatively
class Customer extends \yii\db\ActiveRecord
{
    public function getComments()
    {
        return $this->hasMany(Comment::className(), ['customer_id' => 'id']);
    }
}

$customers = Customer::find()->joinWith([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```

> Info: In Yii 1.1, there is a concept called *scope*. Scope is no longer directly supported in Yii 2.0, and you should use customized query classes and query methods to achieve the same goal.

Selecting extra fields

When Active Record instance is populated from query results, its attributes are filled up by corresponding column values from received data set.

You are able to fetch additional columns or values from query and store it inside the Active Record. For example, assume we have a table named `room`, which contains information about rooms available in the hotel. Each room stores information about its geometrical size using fields `length`, `width`, `height`. Imagine we need to retrieve list of all available rooms with their volume in descendant order. So you can not calculate volume using PHP, because we need to sort the records by its value, but you also want `volume` to be displayed in the list. To achieve the goal, you need to declare an extra field in your `Room` Active Record class, which will store `volume` value:

```
class Room extends \yii\db\ActiveRecord
{
    public $volume;

    // ...
}
```

Then you need to compose a query, which calculates volume of the room and performs the sort:

```
$rooms = Room::find()
    ->select([
        '{{room}}.*', // select all columns
        '([[length]] * [[width]] * [[height]]) AS volume', // calculate a volume
    ])
    ->orderBy('volume DESC') // apply sort
    ->all();

foreach ($rooms as $room) {
    echo $room->volume; // contains value calculated by SQL
}
```

Ability to select extra fields can be exceptionally useful for aggregation queries. Assume you need to display a list of customers with the count of orders they have made. First of all, you need to declare a `Customer` class with `orders` relation and extra field for count storage:

```
class Customer extends \yii\db\ActiveRecord
{
    public $ordersCount;

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}
```

Then you can compose a query, which joins the orders and calculates their count:

```
$customers = Customer::find()
    ->select([
        '{{customer}}.*', // select all customer fields
        'COUNT({{order}}.id) AS ordersCount' // calculate orders count
    ])
    ->joinWith('orders') // ensure table junction
    ->groupBy('{{customer}}.id') // group the result to ensure aggregation function wo
rks
    ->all();
```

A disadvantage of using this method would be that, if the information isn't loaded on the SQL query - it has to be calculated separately. Thus, if you have found particular record via regular query without extra select statements, it will be unable to return actual value for the extra field. Same will happen for the newly saved record.

```
$room = new Room();
$room->length = 100;
$room->width = 50;
$room->height = 2;

$room->volume; // this value will be `null`, since it was not declared yet
```

Using the [[yii\db\BaseActiveRecord::**get()**|get()]] and [[yii\db\BaseActiveRecord::**set()**|set()]] magic methods we can emulate the behavior of a property:

```
class Room extends \yii\db\ActiveRecord
{
    private $_volume;

    public function setVolume($volume)
    {
        $this->_volume = (float) $volume;
    }

    public function getVolume()
    {
        if (empty($this->length) || empty($this->width) || empty($this->height)) {
            return null;
        }

        if ($this->_volume === null) {
            $this->setVolume(
                $this->length * $this->width * $this->height
            );
        }

        return $this->_volume;
    }

    // ...
}
```

When the select query doesn't provide the volume, the model will be able to calculate it automatically using the attributes of the model.

You can calculate the aggregation fields as well using defined relations:

```
class Customer extends \yii\db\ActiveRecord
{
    private $_ordersCount;

    public function setOrdersCount($count)
    {
        $this->_ordersCount = (int) $count;
    }

    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // this avoid calling a query searching for null primary keys
        }

        if ($this->_ordersCount === null) {
            $this->setOrdersCount($this->getOrders()->count()); // calculate aggregati
on on demand from relation
        }

        return $this->_ordersCount;
    }

    // ...

    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }
}
```

With this code, in case 'ordersCount' is present in 'select' statement - `Customer::ordersCount` will be populated by query results, otherwise it will be calculated on demand using `Customer::orders` relation.

This approach can be as well used for creation of the shortcuts for some relational data, especially for the aggregation. For example:

```
class Customer extends \yii\db\ActiveRecord
{
    /**
     * Defines read-only virtual property for aggregation data.
     */
    public function getOrdersCount()
    {
        if ($this->isNewRecord) {
            return null; // this avoid calling a query searching for null primary keys
        }

        return empty($this->ordersAggregation) ? 0 : $this->ordersAggregation[0]['coun
ted'];
    }

    /**
     * Declares normal 'orders' relation.
     */
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id']);
    }

    /**
     * Declares new relation based on 'orders', which provides aggregation.
     */
    public function getOrdersAggregation()
    {
        return $this->getOrders()
            ->select(['customer_id', 'counted' => 'count(*)'])
            ->groupBy('customer_id')
            ->asArray(true);
    }

    // ...
}

foreach (Customer::find()->with('ordersAggregation')->all() as $customer) {
    echo $customer->ordersCount; // outputs aggregation data from relation without ext
ra query due to eager loading
}

$customer = Customer::findOne($pk);
$customer->ordersCount; // output aggregation data from lazy loaded relation
```

Database Migration

During the course of developing and maintaining a database-driven application, the structure of the database being used evolves just like the source code does. For example, during the development of an application, a new table may be found necessary; after the application is deployed to production, it may be discovered that an index should be created to improve the query performance; and so on. Because a database structure change often requires some source code changes, Yii supports the so-called *database migration* feature that allows you to keep track of database changes in terms of *database migrations* which are version-controlled together with the source code.

The following steps show how database migration can be used by a team during development:

1. Tim creates a new migration (e.g. creates a new table, changes a column definition, etc.).
2. Tim commits the new migration into the source control system (e.g. Git, Mercurial).
3. Doug updates his repository from the source control system and receives the new migration.
4. Doug applies the migration to his local development database, thereby synchronizing his database to reflect the changes that Tim has made.

And the following steps show how to deploy a new release with database migrations to production:

1. Scott creates a release tag for the project repository that contains some new database migrations.
2. Scott updates the source code on the production server to the release tag.
3. Scott applies any accumulated database migrations to the production database.

Yii provides a set of migration command line tools that allow you to:

- create new migrations;
- apply migrations;
- revert migrations;
- re-apply migrations;
- show migration history and status.

All these tools are accessible through the command `yii migrate`. In this section we will describe in detail how to accomplish various tasks using these tools. You may also get the usage of each tool via the help command `yii help migrate`.

> Tip: migrations could affect not only database schema but adjust existing data to fit new schema, create RBAC hierarchy or clean up cache.

Creating Migrations

To create a new migration, run the following command:

```
yii migrate/create <name>
```

The required `name` argument gives a brief description about the new migration. For example, if the migration is about creating a new table named *news*, you may use the name `create_news_table` and run the following command:

```
yii migrate/create create_news_table
```

> Note: Because the `name` argument will be used as part of the generated migration class name, it should only contain letters, digits, and/or underscore characters.

The above command will create a new PHP class file named `m150101_185401_create_news_table.php` in the `@app/migrations` directory. The file contains the following code which mainly declares a migration class `m150101_185401_create_news_table` with the skeleton code:

```php
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {

    }

    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";

        return false;
    }

    /*
    // Use safeUp/safeDown to run migration code within a transaction
    public function safeUp()
    {
    }

    public function safeDown()
    {
    }
    */
}
```

Each database migration is defined as a PHP class extending from [[yii\db\Migration]]. The migration class name is automatically generated in the format of `m<YYMMDD_HHMMSS>_<Name>`, where

- `<YYMMDD_HHMMSS>` refers to the UTC datetime at which the migration creation command is executed.
- `<Name>` is the same as the value of the `name` argument that you provide to the command.

In the migration class, you are expected to write code in the `up()` method that makes changes to the database structure. You may also want to write code in the `down()` method to revert the changes made by `up()`. The `up()` method is invoked when you upgrade the database with this migration, while the `down()` method is invoked when you downgrade the database. The following code shows how you may implement the migration class to create a `news` table:

```php
<?php

use yii\db\Schema;
use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => Schema::TYPE_PK,
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

> Info: Not all migrations are reversible. For example, if the `up()` method deletes a row of a table, you may not be able to recover this row in the `down()` method. Sometimes, you may be just too lazy to implement the `down()`, because it is not very common to revert database migrations. In this case, you should return `false` in the `down()` method to indicate that the migration is not reversible.

The base migration class [[yii\db\Migration]] exposes a database connection via the [[yii\db\Migration::db|db]] property. You can use it to manipulate the database schema using the methods as described in Working with Database Schema.

Rather than using physical types, when creating a table or column you should use *abstract types* so that your migrations are independent of specific DBMS. The [[yii\db\Schema]] class defines a set of constants to represent the supported abstract types. These constants are named in the format of `TYPE_<Name>`. For example, `TYPE_PK` refers to auto-incremental primary key type; `TYPE_STRING` refers to a string type. When a migration is applied to a particular database, the abstract types will be translated into the corresponding physical types. In the case of MySQL, `TYPE_PK` will be turned into `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY`, while `TYPE_STRING` becomes `varchar(255)`.

You can append additional constraints when using abstract types. In the above example, `NOT NULL` is appended to `Schema::TYPE_STRING` to specify that the column cannot be `null`.

> Info: The mapping between abstract types and physical types is specified by the [[yii\db\QueryBuilder::$typeMap|$typeMap]] property in each concrete `QueryBuilder` class.

Since version 2.0.6, you can make use of the newly introduced schema builder which provides more convenient way of defining column schema. So the migration above could be written like the following:

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }
}
```

A list of all available methods for defining the column types is available in the API documentation of [[yii\db\SchemaBuilderTrait]].

Generating Migrations

Since version 2.0.7 migration console provides a convenient way to create migrations.

If the migration name is of a special form, for example `create_xxx_table` or `drop_xxx_table` then the generated migration file will contain extra code, in this case for creating/dropping tables. In the following all variants of this feature are described.

Create Table

```
yii migrate/create create_post_table
```

generates

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey()
        ]);
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

To create table fields right away, specify them via `--fields` option.

```
yii migrate/create create_post_table --fields="title:string,body:text"
```

generates

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

You can specify more field parameters.

```
yii migrate/create create_post_table --fields="title:string(12):notNull:unique,body:te
xt"
```

generates

```
/**
 * Handles the creation for table `post`.
 */
class m150811_220037_create_post_table extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        $this->dropTable('post');
    }
}
```

> Note: primary key is added automatically and is named `id` by default. If you want to use another name you may specify it explicitly like `--fields="name:primaryKey"`.

Foreign keys

Since 2.0.8 the generator supports foreign keys using the `foreignKey` keyword.

```
yii migrate/create create_post_table --fields="author_id:integer:notNull:foreignKey(us
er),category_id:integer:defaultValue(1):foreignKey,title:string,body:text"
```

generates

```
/**
 * Handles the creation for table `post`.
 * Has foreign keys to the tables:
 *
 * - `user`
 * - `category`
 */
class m160328_040430_create_post_table extends Migration
{
    /**
```

```
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'author_id' => $this->integer()->notNull(),
            'category_id' => $this->integer()->defaultValue(1),
            'title' => $this->string(),
            'body' => $this->text(),
        ]);

        // creates index for column `author_id`
        $this->createIndex(
            'idx-post-author_id',
            'post',
            'author_id'
        );

        // add foreign key for table `user`
        $this->addForeignKey(
            'fk-post-author_id',
            'post',
            'author_id',
            'user',
            'id',
            'CASCADE'
        );

        // creates index for column `category_id`
        $this->createIndex(
            'idx-post-category_id',
            'post',
            'category_id'
        );

        // add foreign key for table `category`
        $this->addForeignKey(
            'fk-post-category_id',
            'post',
            'category_id',
            'category',
            'id',
            'CASCADE'
        );
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        // drops foreign key for table `user`
```

```
        $this->dropForeignKey(
            'fk-post-author_id',
            'post'
        );

        // drops index for column `author_id`
        $this->dropIndex(
            'idx-post-author_id',
            'post'
        );

        // drops foreign key for table `category`
        $this->dropForeignKey(
            'fk-post-category_id',
            'post'
        );

        // drops index for column `category_id`
        $this->dropIndex(
            'idx-post-category_id',
            'post'
        );

        $this->dropTable('post');
    }
}
```

The position of the `foreignKey` keyword in the column description doesn't change the generated code. That means:

- `author_id:integer:notNull:foreignKey(user)`
- `author_id:integer:foreignKey(user):notNull`
- `author_id:foreignKey(user):integer:notNull`

All generate the same code.

The `foreignKey` keyword can take a parameter between parenthesis which will be the name of the related table for the generated foreign key. If no parameter is passed then the table name will be deduced from the column name.

In the example above `author_id:integer:notNull:foreignKey(user)` will generate a column named `author_id` with a foreign key to the `user` table while `category_id:integer:defaultValue(1):foreignKey` will generate a column `category_id` with a foreign key to the `category` table.

Since 2.0.11, `foreignKey` keyword accepts a second parameter, separated by whitespace. It accepts the name of the related column for the foreign key generated. If no second parameter is passed, the column name will be fetched from table schema. If no schema

exists, primary key isn't set or is composite, default name `id` will be used.

Drop Table

```
yii migrate/create drop_post_table --fields="title:string(12):notNull:unique,body:text
"
```

generates

```
class m150811_220037_drop_post_table extends Migration
{
    public function up()
    {
        $this->dropTable('post');
    }

    public function down()
    {
        $this->createTable('post', [
            'id' => $this->primaryKey(),
            'title' => $this->string(12)->notNull()->unique(),
            'body' => $this->text()
        ]);
    }
}
```

Add Column

If the migration name is of the form `add_xxx_column_to_yyy_table` then the file content would contain `addColumn` and `dropColumn` statements necessary.

To add column:

```
yii migrate/create add_position_column_to_post_table --fields="position:integer"
```

generates

```
class m150811_220037_add_position_column_to_post_table extends Migration
{
    public function up()
    {
        $this->addColumn('post', 'position', $this->integer());
    }

    public function down()
    {
        $this->dropColumn('post', 'position');
    }
}
```

You can specify multiple columns as follows:

```
yii migrate/create add_xxx_column_yyy_column_to_zzz_table --fields="xxx:integer,yyy:te
xt"
```

Drop Column

If the migration name is of the form `drop_xxx_column_from_yyy_table` then the file content would contain `addColumn` and `dropColumn` statements necessary.

```
yii migrate/create drop_position_column_from_post_table --fields="position:integer"
```

generates

```
class m150811_220037_drop_position_column_from_post_table extends Migration
{
    public function up()
    {
        $this->dropColumn('post', 'position');
    }

    public function down()
    {
        $this->addColumn('post', 'position', $this->integer());
    }
}
```

Add Junction Table

If the migration name is of the form `create_junction_table_for_xxx_and_yyy_tables` or `create_junction_xxx_and_yyy_tables` then code necessary to create junction table will be generated.

```
yii migrate/create create_junction_table_for_post_and_tag_tables --fields="created_at:
dateTime"
```

generates

```
/**
 * Handles the creation for table `post_tag`.
 * Has foreign keys to the tables:
 *
 * - `post`
 * - `tag`
 */
class m160328_041642_create_junction_table_for_post_and_tag_tables extends Migration
{
    /**
     * @inheritdoc
     */
    public function up()
    {
        $this->createTable('post_tag', [
            'post_id' => $this->integer(),
            'tag_id' => $this->integer(),
            'created_at' => $this->dateTime(),
            'PRIMARY KEY(post_id, tag_id)',
        ]);

        // creates index for column `post_id`
        $this->createIndex(
            'idx-post_tag-post_id',
            'post_tag',
            'post_id'
        );

        // add foreign key for table `post`
        $this->addForeignKey(
            'fk-post_tag-post_id',
            'post_tag',
            'post_id',
            'post',
            'id',
            'CASCADE'
        );

        // creates index for column `tag_id`
        $this->createIndex(
            'idx-post_tag-tag_id',
```

```
            'post_tag',
            'tag_id'
        );

        // add foreign key for table `tag`
        $this->addForeignKey(
            'fk-post_tag-tag_id',
            'post_tag',
            'tag_id',
            'tag',
            'id',
            'CASCADE'
        );
    }

    /**
     * @inheritdoc
     */
    public function down()
    {
        // drops foreign key for table `post`
        $this->dropForeignKey(
            'fk-post_tag-post_id',
            'post_tag'
        );

        // drops index for column `post_id`
        $this->dropIndex(
            'idx-post_tag-post_id',
            'post_tag'
        );

        // drops foreign key for table `tag`
        $this->dropForeignKey(
            'fk-post_tag-tag_id',
            'post_tag'
        );

        // drops index for column `tag_id`
        $this->dropIndex(
            'idx-post_tag-tag_id',
            'post_tag'
        );

        $this->dropTable('post_tag');
    }
}
```

Since 2.0.11 foreign key column names for junction tables are fetched from table schema. In case table isn't defined in schema, or the primary key isn't set or is composite, default name `id` is used.

Transactional Migrations

While performing complex DB migrations, it is important to ensure each migration to either succeed or fail as a whole so that the database can maintain integrity and consistency. To achieve this goal, it is recommended that you enclose the DB operations of each migration in a transaction.

An even easier way of implementing transactional migrations is to put migration code in the `safeUp()` and `safeDown()` methods. These two methods differ from `up()` and `down()` in that they are enclosed implicitly in a transaction. As a result, if any operation in these methods fails, all prior operations will be rolled back automatically.

In the following example, besides creating the `news` table we also insert an initial row into this table.

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => $this->primaryKey(),
            'title' => $this->string()->notNull(),
            'content' => $this->text(),
        ]);

        $this->insert('news', [
            'title' => 'test 1',
            'content' => 'content 1',
        ]);
    }

    public function safeDown()
    {
        $this->delete('news', ['id' => 1]);
        $this->dropTable('news');
    }
}
```

Note that usually when you perform multiple DB operations in `safeUp()`, you should reverse their execution order in `safeDown()`. In the above example we first create the table and then insert a row in `safeUp()`; while in `safeDown()` we first delete the row and then drop the table.

> Note: Not all DBMS support transactions. And some DB queries cannot be put into a transaction. For some examples, please refer to implicit commit. If this is the case, you should still implement `up()` and `down()`, instead.

Database Accessing Methods

The base migration class [[yii\db\Migration]] provides a set of methods to let you access and manipulate databases. You may find these methods are named similarly as the DAO methods provided by the [[yii\db\Command]] class. For example, the [[yii\db\Migration::createTable()]] method allows you to create a new table, just like [[yii\db\Command::createTable()]] does.

The benefit of using the methods provided by [[yii\db\Migration]] is that you do not need to explicitly create [[yii\db\Command]] instances and the execution of each method will automatically display useful messages telling you what database operations are done and how long they take.

Below is the list of all these database accessing methods:

- [[yii\db\Migration::execute()|execute()]]: executing a SQL statement
- [[yii\db\Migration::insert()|insert()]]: inserting a single row
- [[yii\db\Migration::batchInsert()|batchInsert()]]: inserting multiple rows
- [[yii\db\Migration::update()|update()]]: updating rows
- [[yii\db\Migration::delete()|delete()]]: deleting rows
- [[yii\db\Migration::createTable()|createTable()]]: creating a table
- [[yii\db\Migration::renameTable()|renameTable()]]: renaming a table
- [[yii\db\Migration::dropTable()|dropTable()]]: removing a table
- [[yii\db\Migration::truncateTable()|truncateTable()]]: removing all rows in a table
- [[yii\db\Migration::addColumn()|addColumn()]]: adding a column
- [[yii\db\Migration::renameColumn()|renameColumn()]]: renaming a column
- [[yii\db\Migration::dropColumn()|dropColumn()]]: removing a column
- [[yii\db\Migration::alterColumn()|alterColumn()]]: altering a column
- [[yii\db\Migration::addPrimaryKey()|addPrimaryKey()]]: adding a primary key
- [[yii\db\Migration::dropPrimaryKey()|dropPrimaryKey()]]: removing a primary key
- [[yii\db\Migration::addForeignKey()|addForeignKey()]]: adding a foreign key
- [[yii\db\Migration::dropForeignKey()|dropForeignKey()]]: removing a foreign key
- [[yii\db\Migration::createIndex()|createIndex()]]: creating an index
- [[yii\db\Migration::dropIndex()|dropIndex()]]: removing an index
- [[yii\db\Migration::addCommentOnColumn()|addCommentOnColumn()]]: adding comment to column
- [[yii\db\Migration::dropCommentFromColumn()|dropCommentFromColumn()]]: dropping comment from column

- [[yii\db\Migration::addCommentOnTable()|addCommentOnTable()]]: adding comment to table
- [[yii\db\Migration::dropCommentFromTable()|dropCommentFromTable()]]: dropping comment from table

> Info: [[yii\db\Migration]] does not provide a database query method. This is because you normally do not need to display extra message about retrieving data from a database. It is also because you can use the powerful Query Builder to build and run complex queries.
>
> Note: When manipulating data using a migration you may find that using your Active Record classes for this might be useful because some of the logic is already implemented there. Keep in mind however, that in contrast to code written in the migrations, who's nature is to stay constant forever, application logic is subject to change. So when using Active Record in migration code, changes to the logic in the Active Record layer may accidentally break existing migrations. For this reason migration code should be kept independent from other application logic such as Active Record classes.

Applying Migrations

To upgrade a database to its latest structure, you should apply all available new migrations using the following command:

```
yii migrate
```

This command will list all migrations that have not been applied so far. If you confirm that you want to apply these migrations, it will run the `up()` or `safeUp()` method in every new migration class, one after another, in the order of their timestamp values. If any of the migrations fails, the command will quit without applying the rest of the migrations.

> Tip: In case you don't have command line at your server you may try web shell extension.

For each migration that has been successfully applied, the command will insert a row into a database table named `migration` to record the successful application of the migration. This will allow the migration tool to identify which migrations have been applied and which have not.

> Info: The migration tool will automatically create the `migration` table in the database specified by the [[yii\console\controllers\MigrateController::db|db]] option of the command. By default, the database is specified by the `db` application component.

Sometimes, you may only want to apply one or a few new migrations, instead of all available migrations. You can do so by specifying the number of migrations that you want to apply when running the command. For example, the following command will try to apply the next three available migrations:

```
yii migrate 3
```

You can also explicitly specify a particular migration to which the database should be migrated by using the `migrate/to` command in one of the following formats:

```
yii migrate/to 150101_185401                      # using timestamp to specify the mig
ration
yii migrate/to "2015-01-01 18:54:01"              # using a string that can be parsed
by strtotime()
yii migrate/to m150101_185401_create_news_table   # using full name
yii migrate/to 1392853618                         # using UNIX timestamp
```

If there are any unapplied migrations earlier than the specified one, they will all be applied before the specified migration is applied.

If the specified migration has already been applied before, any later applied migrations will be reverted.

Reverting Migrations

To revert (undo) one or multiple migrations that have been applied before, you can run the following command:

```
yii migrate/down     # revert the most recently applied migration
yii migrate/down 3   # revert the most 3 recently applied migrations
```

> Note: Not all migrations are reversible. Trying to revert such migrations will cause an error and stop the entire reverting process.

Redoing Migrations

Redoing migrations means first reverting the specified migrations and then applying again. This can be done as follows:

```
yii migrate/redo        # redo the last applied migration
yii migrate/redo 3      # redo the last 3 applied migrations
```

> Note: If a migration is not reversible, you will not be able to redo it.

Listing Migrations

To list which migrations have been applied and which are not, you may use the following commands:

```
yii migrate/history     # showing the last 10 applied migrations
yii migrate/history 5   # showing the last 5 applied migrations
yii migrate/history all # showing all applied migrations

yii migrate/new         # showing the first 10 new migrations
yii migrate/new 5       # showing the first 5 new migrations
yii migrate/new all     # showing all new migrations
```

Modifying Migration History

Instead of actually applying or reverting migrations, sometimes you may simply want to mark that your database has been upgraded to a particular migration. This often happens when you manually change the database to a particular state and you do not want the migration(s) for that change to be re-applied later. You can achieve this goal with the following command:

```
yii migrate/mark 150101_185401                      # using timestamp to specify the m
igration
yii migrate/mark "2015-01-01 18:54:01"              # using a string that can be parse
d by strtotime()
yii migrate/mark m150101_185401_create_news_table   # using full name
yii migrate/mark 1392853618                         # using UNIX timestamp
```

The command will modify the `migration` table by adding or deleting certain rows to indicate that the database has been applied migrations to the specified one. No migrations will be applied or reverted by this command.

Customizing Migrations

There are several ways to customize the migration command.

Using Command Line Options

The migration command comes with a few command-line options that can be used to customize its behaviors:

- `interactive` : boolean (defaults to `true`), specifies whether to perform migrations in an interactive mode. When this is `true` , the user will be prompted before the command performs certain actions. You may want to set this to `false` if the command is being used in a background process.
- `migrationPath` : string (defaults to `@app/migrations`), specifies the directory storing all migration class files. This can be specified as either a directory path or a path alias. Note that the directory must exist, or the command may trigger an error.
- `migrationTable` : string (defaults to `migration`), specifies the name of the database table for storing migration history information. The table will be automatically created by the command if it does not exist. You may also manually create it using the structure `version varchar(255) primary key, apply_time integer` .
- `db` : string (defaults to `db`), specifies the ID of the database application component. It represents the database that will be migrated using this command.
- `templateFile` : string (defaults to `@yii/views/migration.php`), specifies the path of the template file that is used for generating skeleton migration class files. This can be specified as either a file path or a path alias. The template file is a PHP script in which you can use a predefined variable named `$className` to get the migration class name.
- `generatorTemplateFiles` : array (defaults to `[

  ```
  'create_table' => '@yii/views/createTableMigration.php',
  'drop_table' => '@yii/views/dropTableMigration.php',
  'add_column' => '@yii/views/addColumnMigration.php',
  'drop_column' => '@yii/views/dropColumnMigration.php',
  'create_junction' => '@yii/views/createTableMigration.php'
  ```

  ]`), specifies template files for generating migration code. See "Generating Migrations" for more details.
- `fields` : array of column definition strings used for creating migration code. Defaults to `[]` . The format of each definition is `COLUMN_NAME:COLUMN_TYPE:COLUMN_DECORATOR` . For example, `--fields=name:string(12):notNull` produces a string column of size 12 which is not `null` .

The following example shows how you can use these options.

For example, if we want to migrate a `forum` module whose migration files are located within the module's `migrations` directory, we can use the following command:

```
# migrate the migrations in a forum module non-interactively
yii migrate --migrationPath=@app/modules/forum/migrations --interactive=0
```

Configuring Command Globally

Instead of entering the same option values every time you run the migration command, you may configure it once for all in the application configuration like shown below:

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationTable' => 'backend_migration',
        ],
    ],
];
```

With the above configuration, each time you run the migration command, the `backend_migration` table will be used to record the migration history. You no longer need to specify it via the `migrationTable` command-line option.

Namespaced Migrations

Since 2.0.10 you can use namespaces for the migration classes. You can specify the list of the migration namespaces via [[yii\console\controllers\MigrateController::migrationNamespaces|migrationNamespaces]]. Using of the namespaces for migration classes allows you usage of the several source locations for the migrations. For example:

```
return [
    'controllerMap' => [
        'migrate' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => [
                'app\migrations', // Common migrations for the whole application
                'module\migrations', // Migrations for the specific project's module
                'some\extension\migrations', // Migrations for the specific extension
            ],
        ],
    ],
];
```

> Note: migrations applied from different namespaces will create a **single** migration history, e.g. you might be unable to apply or revert migrations from particular namespace only.

While operating namespaced migrations: creating new, reverting and so on, you should specify full namespace before migration name. Note that backslash (`\`) symbol is usually considered a special character in the shell, so you need to escape it properly to avoid shell errors or incorrect behavior. For example:

```
yii migrate/create 'app\\migrations\\createUserTable'
```

> Note: migrations specified via [[yii\console\controllers\MigrateController::migrationPath|migrationPath]] can not contain a namespace, namespaced migration can be applied only via [[yii\console\controllers\MigrateController::migrationNamespaces]] property.

Separated Migrations

Sometimes using single migration history for all project migrations is not desirable. For example: you may install some 'blog' extension, which contains fully separated functionality and contain its own migrations, which should not affect the ones dedicated to main project functionality.

If you want several migrations to be applied and tracked down completely separated from each other, you can configure multiple migration commands which will use different namespaces and migration history tables:

```
return [
    'controllerMap' => [
        // Common migrations for the whole application
        'migrate-app' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['app\migrations'],
            'migrationTable' => 'migration_app',
        ],
        // Migrations for the specific project's module
        'migrate-module' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationNamespaces' => ['module\migrations'],
            'migrationTable' => 'migration_module',
        ],
        // Migrations for the specific extension
        'migrate-rbac' => [
            'class' => 'yii\console\controllers\MigrateController',
            'migrationPath' => '@yii/rbac/migrations',
            'migrationTable' => 'migration_rbac',
        ],
    ],
];
```

Note that to synchronize database you now need to run multiple commands instead of one:

```
yii migrate-app
yii migrate-module
yii migrate-rbac
```

Migrating Multiple Databases

By default, migrations are applied to the same database specified by the `db` application component. If you want them to be applied to a different database, you may specify the `db` command-line option like shown below,

```
yii migrate --db=db2
```

The above command will apply migrations to the `db2` database.

Sometimes it may happen that you want to apply *some* of the migrations to one database, while some others to another database. To achieve this goal, when implementing a migration class you should explicitly specify the DB component ID that the migration would use, like the following:

```
<?php

use yii\db\Migration;

class m150101_185401_create_news_table extends Migration
{
    public function init()
    {
        $this->db = 'db2';
        parent::init();
    }
}
```

The above migration will be applied to `db2`, even if you specify a different database through the `db` command-line option. Note that the migration history will still be recorded in the database specified by the `db` command-line option.

If you have multiple migrations that use the same database, it is recommended that you create a base migration class with the above `init()` code. Then each migration class can extend from this base class.

> Tip: Besides setting the [[yii\db\Migration::db|db]] property, you can also operate on different databases by creating new database connections to them in your migration classes. You then use the DAO methods with these connections to manipulate different databases.

Another strategy that you can take to migrate multiple databases is to keep migrations for different databases in different migration paths. Then you can migrate these databases in separate commands like the following:

```
yii migrate --migrationPath=@app/migrations/db1 --db=db1
yii migrate --migrationPath=@app/migrations/db2 --db=db2
...
```

The first command will apply migrations in `@app/migrations/db1` to the `db1` database, the second command will apply migrations in `@app/migrations/db2` to `db2`, and so on.

Creating Forms

ActiveRecord based forms: ActiveForm

The primary way of using forms in Yii is through [[yii\widgets\ActiveForm]]. This approach should be preferred when the form is based upon a model. Additionally, there are some useful methods in [[yii\helpers\Html]] that are typically used for adding buttons and help text to any form.

A form, that is displayed on the client-side, will in most cases have a corresponding model which is used to validate its input on the server-side (Check the Validating Input section for more details on validation). When creating model-based forms, the first step is to define the model itself. The model can be either based upon an Active Record class, representing some data from the database, or a generic Model class (extending from [[yii\base\Model]]) to capture arbitrary input, for example a login form. In the following example, we show how a generic model can be used for a login form:

```
<?php

class LoginForm extends \yii\base\Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // define validation rules here
        ];
    }
}
```

In the controller, we will pass an instance of that model to the view, wherein the [[yii\widgets\ActiveForm|ActiveForm]] widget is used to display the form:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'login-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>

    <div class="form-group">
        <div class="col-lg-offset-1 col-lg-11">
            <?= Html::submitButton('Login', ['class' => 'btn btn-primary']) ?>
        </div>
    </div>
<?php ActiveForm::end() ?>
```

Wrapping with `begin()` and `end()`

In the above code, [[yii\widgets\ActiveForm::begin()|ActiveForm::begin()]] not only creates a form instance, but also marks the beginning of the form. All of the content placed between [[yii\widgets\ActiveForm::begin()|ActiveForm::begin()]] and [[yii\widgets\ActiveForm::end()|ActiveForm::end()]] will be wrapped within the HTML `<form>` tag. As with any widget, you can specify some options as to how the widget should be configured by passing an array to the `begin` method. In this case, an extra CSS class and identifying ID are passed to be used in the opening `<form>` tag. For all available options, please refer to the API documentation of [[yii\widgets\ActiveForm]].

ActiveField .

In order to create a form element in the form, along with the element's label, and any applicable JavaScript validation, the [[yii\widgets\ActiveForm::field()|ActiveForm::field()]] method is called, which returns an instance of [[yii\widgets\ActiveField]]. When the result of this method is echoed directly, the result is a regular (text) input. To customize the output, you can chain additional methods of [[yii\widgets\ActiveField|ActiveField]] to this call:

```
// a password input
<?= $form->field($model, 'password')->passwordInput() ?>
// adding a hint and a customized label
<?= $form->field($model, 'username')->textInput()->hint('Please enter your name')->lab
el('Name') ?>
// creating a HTML5 email input element
<?= $form->field($model, 'email')->input('email') ?>
```

This will create all the `<label>`, `<input>` and other tags according to the [[yii\widgets\ActiveField::$template|template]] defined by the form field. The name of the input field is determined automatically from the model's [[yii\base\Model::formName()|form name]] and the attribute name. For example, the name for the input field for the `username` attribute in the above example will be `LoginForm[username]`. This naming rule will result in an array of all attributes for the login form to be available in `$_POST['LoginForm']` on the server-side.

> Tip: If you have only one model in a form and want to simplify the input names you may skip the array part by overriding the [[yii\base\Model::formName()|formName()]] method of the model to return an empty string. This can be useful for filter models used in the GridView to create nicer URLs.

Specifying the attribute of the model can be done in more sophisticated ways. For example when an attribute may take an array value when uploading multiple files or selecting multiple items you may specify it by appending `[]` to the attribute name:

```
// allow multiple files to be uploaded:
echo $form->field($model, 'uploadFile[]')->fileInput(['multiple'=>'multiple']);

// allow multiple items to be checked:
echo $form->field($model, 'items[]')->checkboxList(['a' => 'Item A', 'b' => 'Item B',
'c' => 'Item C']);
```

Be careful when naming form elements such as submit buttons. According to the jQuery documentation there are some reserved names that can cause conflicts:

> Forms and their child elements should not use input names or ids that conflict with properties of a form, such as `submit`, `length`, or `method`. Name conflicts can cause confusing failures. For a complete list of rules and to check your markup for these problems, see DOMLint.

Additional HTML tags can be added to the form using plain HTML or using the methods from the [[yii\helpers\Html|Html]]-helper class like it is done in the above example with [[yii\helpers\Html::submitButton()|Html::submitButton()]].

> Tip: If you are using Twitter Bootstrap CSS in your application you may want to use [[yii\bootstrap\ActiveForm]] instead of [[yii\widgets\ActiveForm]]. The former extends from the latter and uses Bootstrap-specific styles when generating form input fields.
>
> Tip: In order to style required fields with asterisks, you can use the following CSS:
>
> ```
> div.required label.control-label:after {
>     content: " *";
>     color: red;
> }
> ```

Creating Lists

There are 3 types of lists:

- Dropdown lists
- Radio lists
- Checkbox lists

To create a list, you have to prepare the items. This can be done manually:

```
$items = [
    1 => 'item 1',
    2 => 'item 2'
]
```

or by retrieval from the DB:

```
$items = Category::find()
        ->select(['id', 'label'])
        ->indexBy('id')
        ->column();
```

These `$items` have to be processed by the different list widgets. The value of the form field (and the current active item) will be automatically set by the current value of the `$model` 's attribute.

Creating a drop-down list

We can use ActiveField [[\yii\widgets\ActiveField::dropDownList()]] method to create a drop-down list:

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->dropdownList([
        1 => 'item 1',
        2 => 'item 2'
    ],
    ['prompt'=>'Select Category']
);
```

Creating a radio list

We can use ActiveField [[\yii\widgets\ActiveField::radioList()]] method to create a radio list:

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->radioList([
    1 => 'radio 1',
    2 => 'radio 2'
]);
```

Creating a checkbox List

We can use ActiveField [[\yii\widgets\ActiveField::checkboxList()]] method to create a checkbox list:

```
/* @var $form yii\widgets\ActiveForm */

echo $form->field($model, 'category')->checkboxList([
    1 => 'checkbox 1',
    2 => 'checkbox 2'
]);
```

Working with Pjax

The [[yii\widgets\Pjax|Pjax]] widget allows you to update a certain section of a page instead of reloading the entire page. You can use it to update only the form and replace its contents after the submission.

You can configure [[yii\widgets\Pjax::$formSelector|$formSelector]] to specify which form submission may trigger pjax. If not set, all forms with `data-pjax` attribute within the enclosed content of Pjax will trigger pjax requests.

```
use yii\widgets\Pjax;
use yii\widgets\ActiveForm;

Pjax::begin([
    // Pjax options
]);
    $form = ActiveForm::begin([
        'options' => ['data' => ['pjax' => true]],
        // more ActiveForm options
    ]);

        // ActiveForm content

    ActiveForm::end();
Pjax::end();
```

> Tip: Be careful with the links inside the [[yii\widgets\Pjax|Pjax]] widget since the response will also be rendered inside the widget. To prevent this, use the `data-pjax="0"` HTML attribute.

Values in Submit Buttons and File Upload

There are known issues using `jQuery.serializeArray()` when dealing with files and submit button values which won't be solved and are instead deprecated in favor of the `FormData` class introduced in HTML5.

That means the only official support for files and submit button values with ajax or using the [[yii\widgets\Pjax|Pjax]] widget depends on the browser support for the `FormData` class.

Further Reading

The next section Validating Input handles the validation of the submitted form data on the server-side as well as ajax and client-side validation.

To read about more complex usage of forms, you may want to check out the following sections:

- Collecting Tabular Input for collecting data for multiple models of the same kind.
- Getting Data for Multiple Models for handling multiple different models in the same form.
- Uploading Files on how to use forms for uploading files.

Validating Input

As a rule of thumb, you should never trust the data received from end users and should always validate it before putting it to good use.

Given a model populated with user inputs, you can validate the inputs by calling the [[yii\base\Model::validate()]] method. The method will return a boolean value indicating whether the validation succeeded or not. If not, you may get the error messages from the [[yii\base\Model::errors]] property. For example,

```
$model = new \app\models\ContactForm();

// populate model attributes with user inputs
$model->load(\Yii::$app->request->post());
// which is equivalent to the following:
// $model->attributes = \Yii::$app->request->post('ContactForm');

if ($model->validate()) {
    // all inputs are valid
} else {
    // validation failed: $errors is an array containing error messages
    $errors = $model->errors;
}
```

Declaring Rules

To make `validate()` really work, you should declare validation rules for the attributes you plan to validate. This should be done by overriding the [[yii\base\Model::rules()]] method. The following example shows how the validation rules for the `ContactForm` model are declared:

```
public function rules()
{
    return [
        // the name, email, subject and body attributes are required
        [['name', 'email', 'subject', 'body'], 'required'],

        // the email attribute should be a valid email address
        ['email', 'email'],
    ];
}
```

The [[yii\base\Model::rules()|rules()]] method should return an array of rules, each of which is an array of the following format:

```
[
    // required, specifies which attributes should be validated by this rule.
    // For a single attribute, you can use the attribute name directly
    // without having it in an array
    ['attribute1', 'attribute2', ...],

    // required, specifies the type of this rule.
    // It can be a class name, validator alias, or a validation method name
    'validator',

    // optional, specifies in which scenario(s) this rule should be applied
    // if not given, it means the rule applies to all scenarios
    // You may also configure the "except" option if you want to apply the rule
    // to all scenarios except the listed ones
    'on' => ['scenario1', 'scenario2', ...],

    // optional, specifies additional configurations for the validator object
    'property1' => 'value1', 'property2' => 'value2', ...
]
```

For each rule you must specify at least which attributes the rule applies to and what is the type of the rule. You can specify the rule type in one of the following forms:

- the alias of a core validator, such as `required`, `in`, `date`, etc. Please refer to the Core Validators for the complete list of core validators.
- the name of a validation method in the model class, or an anonymous function. Please refer to the Inline Validators subsection for more details.
- a fully qualified validator class name. Please refer to the Standalone Validators subsection for more details.

A rule can be used to validate one or multiple attributes, and an attribute may be validated by one or multiple rules. A rule may be applied in certain scenarios only by specifying the `on` option. If you do not specify an `on` option, it means the rule will be applied to all scenarios.

When the `validate()` method is called, it does the following steps to perform validation:

1. Determine which attributes should be validated by getting the attribute list from [[yii\base\Model::scenarios()]] using the current [[yii\base\Model::scenario|scenario]]. These attributes are called *active attributes*.
2. Determine which validation rules should be used by getting the rule list from [[yii\base\Model::rules()]] using the current [[yii\base\Model::scenario|scenario]]. These rules are called *active rules*.

3. Use each active rule to validate each active attribute which is associated with the rule. The validation rules are evaluated in the order they are listed.

According to the above validation steps, an attribute will be validated if and only if it is an active attribute declared in `scenarios()` and is associated with one or multiple active rules declared in `rules()`.

> Note: It is handy to give names to rules i.e.
>
> ```
> public function rules()
> {
>     return [
>         // ...
>         'password' => [['password'], 'string', 'max' => 60],
>     ];
> }
> ```
>
> You can use it in a child model:
>
> ```php public function rules() { $rules = parent::rules(); unset($rules['password']); return $rules; }
> ```

Customizing Error Messages

Most validators have default error messages that will be added to the model being validated when its attributes fail the validation. For example, the [[yii\validators\RequiredValidator|required]] validator will add a message "Username cannot be blank." to a model when the `username` attribute fails the rule using this validator.

You can customize the error message of a rule by specifying the `message` property when declaring the rule, like the following,

```
public function rules()
{
    return [
        ['username', 'required', 'message' => 'Please choose a username.'],
    ];
}
```

Some validators may support additional error messages to more precisely describe different causes of validation failures. For example, the [[yii\validators\NumberValidator|number]] validator supports [[yii\validators\NumberValidator::tooBig|tooBig]] and [[yii\validators\NumberValidator::tooSmall|tooSmall]] to describe the validation failure when the value being validated is too big and too small, respectively. You may configure these error messages like configuring other properties of validators in a validation rule.

Validation Events

When [[yii\base\Model::validate()]] is called, it will call two methods that you may override to customize the validation process:

- [[yii\base\Model::beforeValidate()]]: the default implementation will trigger a [[yii\base\Model::EVENT_BEFORE_VALIDATE]] event. You may either override this method or respond to this event to do some preprocessing work (e.g. normalizing data inputs) before the validation occurs. The method should return a boolean value indicating whether the validation should proceed or not.
- [[yii\base\Model::afterValidate()]]: the default implementation will trigger a [[yii\base\Model::EVENT_AFTER_VALIDATE]] event. You may either override this method or respond to this event to do some postprocessing work after the validation is completed.

Conditional Validation

To validate attributes only when certain conditions apply, e.g. the validation of one attribute depends on the value of another attribute you can use the [[yii\validators\Validator::when|when]] property to define such conditions. For example,

```
    ['state', 'required', 'when' => function($model) {
        return $model->country == 'USA';
    }]
```

The [[yii\validators\Validator::when|when]] property takes a PHP callable with the following signature:

```
/**
 * @param Model $model the model being validated
 * @param string $attribute the attribute being validated
 * @return bool whether the rule should be applied
 */
function ($model, $attribute)
```

If you also need to support client-side conditional validation, you should configure the [[yii\validators\Validator::whenClient|whenClient]] property which takes a string representing a JavaScript function whose return value determines whether to apply the rule or not. For example,

```
    ['state', 'required', 'when' => function ($model) {
        return $model->country == 'USA';
    }, 'whenClient' => "function (attribute, value) {
        return $('#country').val() == 'USA';
    }"]
```

Data Filtering

User inputs often need to be filtered or preprocessed. For example, you may want to trim the spaces around the `username` input. You may use validation rules to achieve this goal.

The following examples shows how to trim the spaces in the inputs and turn empty inputs into nulls by using the trim and default core validators:

```
return [
    [['username', 'email'], 'trim'],
    [['username', 'email'], 'default'],
];
```

You may also use the more general filter validator to perform more complex data filtering.

As you can see, these validation rules do not really validate the inputs. Instead, they will process the values and save them back to the attributes being validated.

Handling Empty Inputs

When input data are submitted from HTML forms, you often need to assign some default values to the inputs if they are empty. You can do so by using the default validator. For example,

```
return [
    // set "username" and "email" as null if they are empty
    [['username', 'email'], 'default'],

    // set "level" to be 1 if it is empty
    ['level', 'default', 'value' => 1],
];
```

By default, an input is considered empty if its value is an empty string, an empty array or a `null`. You may customize the default empty detection logic by configuring the [[yii\validators\Validator::isEmpty]] property with a PHP callable. For example,

```
    ['agree', 'required', 'isEmpty' => function ($value) {
        return empty($value);
    }]
```

> Note: Most validators do not handle empty inputs if their [[yii\validators\Validator::skipOnEmpty]] property takes the default value `true`. They will simply be skipped during validation if their associated attributes receive empty inputs. Among the core validators, only the `captcha`, `default`, `filter`, `required`, and `trim` validators will handle empty inputs.

Ad Hoc Validation

Sometimes you need to do *ad hoc validation* for values that are not bound to any model.

If you only need to perform one type of validation (e.g. validating email addresses), you may call the [[yii\validators\Validator::validate()|validate()]] method of the desired validator, like the following:

```
$email = 'test@example.com';
$validator = new yii\validators\EmailValidator();

if ($validator->validate($email, $error)) {
    echo 'Email is valid.';
} else {
    echo $error;
}
```

> Note: Not all validators support this type of validation. An example is the unique core validator which is designed to work with a model only.

If you need to perform multiple validations against several values, you can use [[yii\base\DynamicModel]] which supports declaring both attributes and rules on the fly. Its usage is like the following:

```
public function actionSearch($name, $email)
{
    $model = DynamicModel::validateData(compact('name', 'email'), [
        [['name', 'email'], 'string', 'max' => 128],
        ['email', 'email'],
    ]);

    if ($model->hasErrors()) {
        // validation fails
    } else {
        // validation succeeds
    }
}
```

The [[yii\base\DynamicModel::validateData()]] method creates an instance of `DynamicModel`, defines the attributes using the given data (`name` and `email` in this example), and then calls [[yii\base\Model::validate()]] with the given rules.

Alternatively, you may use the following more "classic" syntax to perform ad hoc data validation:

```
public function actionSearch($name, $email)
{
    $model = new DynamicModel(compact('name', 'email'));
    $model->addRule(['name', 'email'], 'string', ['max' => 128])
        ->addRule('email', 'email')
        ->validate();

    if ($model->hasErrors()) {
        // validation fails
    } else {
        // validation succeeds
    }
}
```

After validation, you can check if the validation succeeded or not by calling the [[yii\base\DynamicModel::hasErrors()|hasErrors()]] method, and then get the validation errors from the [[yii\base\DynamicModel::errors|errors]] property, like you do with a normal model. You may also access the dynamic attributes defined through the model instance, e.g., `$model->name` and `$model->email`.

Creating Validators

Besides using the core validators included in the Yii releases, you may also create your own validators. You may create inline validators or standalone validators.

Inline Validators

An inline validator is one defined in terms of a model method or an anonymous function. The signature of the method/function is:

```
/**
 * @param string $attribute the attribute currently being validated
 * @param mixed $params the value of the "params" given in the rule
 * @param \yii\validators\InlineValidator related InlineValidator instance.
 * This parameter is available since version 2.0.11.
 */
function ($attribute, $params, $validator)
```

If an attribute fails the validation, the method/function should call [[yii\base\Model::addError()]] to save the error message in the model so that it can be retrieved back later to present to end users.

Below are some examples:

```
use yii\base\Model;

class MyForm extends Model
{
    public $country;
    public $token;

    public function rules()
    {
        return [
            // an inline validator defined as the model method validateCountry()
            ['country', 'validateCountry'],

            // an inline validator defined as an anonymous function
            ['token', function ($attribute, $params, $validator) {
                if (!ctype_alnum($this->$attribute)) {
                    $this->addError($attribute, 'The token must contain letters or dig
its.');
                }
            }],
        ];
    }

    public function validateCountry($attribute, $params, $validator)
    {
        if (!in_array($this->$attribute, ['USA', 'Web'])) {
            $this->addError($attribute, 'The country must be either "USA" or "Web".');
        }
    }
}
```

> Note: Since version 2.0.11 you can use [[yii\validators\InlineValidator::addError()]] for adding errors instead. That way the error message can be formatted using [[yii\i18n\I18N::format()]] right away. Use `{attribute}` and `{value}` in the error message to refer to an attribute label (no need to get it manually) and attribute value accordingly:
>
> ```
> $validator->addError($this, $attribute, 'The value "{value}" is not acceptable fo
> r {attribute}.');
> ```
>
> Note: By default, inline validators will not be applied if their associated attributes receive empty inputs or if they have already failed some validation rules. If you want to make sure a rule is always applied, you may configure the [[yii\validators\Validator::skipOnEmpty|skipOnEmpty]] and/or [[yii\validators\Validator::skipOnError|skipOnError]] properties to be `false` in the rule declarations. For example:
>
> ```
> [
>     ['country', 'validateCountry', 'skipOnEmpty' => false, 'skipOnError' => false
> ],
> ]
> ```

Standalone Validators

A standalone validator is a class extending [[yii\validators\Validator]] or its child class. You may implement its validation logic by overriding the [[yii\validators\Validator::validateAttribute()]] method. If an attribute fails the validation, call [[yii\base\Model::addError()]] to save the error message in the model, like you do with inline validators.

For example the inline validator above could be moved into new [[components/validators/CountryValidator]] class.

```
namespace app\components;

use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Web'])) {
            $this->addError($model, $attribute, 'The country must be either "USA" or "
Web".');
        }
    }
}
```

If you want your validator to support validating a value without a model, you should also override [[yii\validators\Validator::validate()]]. You may also override [[yii\validators\Validator::validateValue()]] instead of `validateAttribute()` and `validate()` because by default the latter two methods are implemented by calling `validateValue()`.

Below is an example of how you could use the above validator class within your model.

```
namespace app\models;

use Yii;
use yii\base\Model;
use app\components\validators\CountryValidator;

class EntryForm extends Model
{
    public $name;
    public $email;
    public $country;

    public function rules()
    {
        return [
            [['name', 'email'], 'required'],
            ['country', CountryValidator::className()],
            ['email', 'email'],
        ];
    }
}
```

Multiple Attributes Validation

Sometimes validators involve multiple attributes. Consider the following form:

```
class MigrationForm extends \yii\base\Model
{
    /**
     * Minimal funds amount for one adult person
     */
    const MIN_ADULT_FUNDS = 3000;
    /**
     * Minimal funds amount for one child
     */
    const MIN_CHILD_FUNDS = 1500;

    public $personalSalary;
    public $spouseSalary;
    public $childrenCount;
    public $description;

    public function rules()
    {
        return [
            [['personalSalary', 'description'], 'required'],
            [['personalSalary', 'spouseSalary'], 'integer', 'min' => self::MIN_ADULT_F
UNDS],
            ['childrenCount', 'integer', 'min' => 0, 'max' => 5],
            [['spouseSalary', 'childrenCount'], 'default', 'value' => 0],
            ['description', 'string'],
        ];
    }
}
```

Creating validator

Let's say we need to check if the family income is enough for children. We can create inline validator `validateChildrenFunds` for that which will run only when `childrenCount` is more than 0.

Note that we can't use all validated attributes (`['personalSalary', 'spouseSalary', 'childrenCount']`) when attaching validator. This is because the same validator will run for each attribute (3 times in total) and we only need to run it once for the whole attribute set.

You can use any of these attributes instead (or use what you think is the most relevant):

```
['childrenCount', 'validateChildrenFunds', 'when' => function ($model) {
    return $model->childrenCount > 0;
}],
```

Implementation of `validateChildrenFunds` can be like this:

```
public function validateChildrenFunds($attribute, $params)
{
    $totalSalary = $this->personalSalary + $this->spouseSalary;
    // Double the minimal adult funds if spouse salary is specified
    $minAdultFunds = $this->spouseSalary ? self::MIN_ADULT_FUNDS * 2 : self::MIN_ADULT
_FUNDS;
    $childFunds = $totalSalary - $minAdultFunds;
    if ($childFunds / $this->childrenCount < self::MIN_CHILD_FUNDS) {
        $this->addError('childrenCount', 'Your salary is not enough for children.');
    }
}
```

You can ignore `$attribute` parameter because validation is not related to just one attribute.

Adding errors

Adding error in case of multiple attributes can vary depending on desired form design:

- Select the most relevant field in your opinion and add error to it's attribute:

```
$this->addError('childrenCount', 'Your salary is not enough for children.');
```

- Select multiple important relevant attributes or all attributes and add the same error message to them. We can store message in separate variable before passing it to `addError` to keep code DRY.

```
$message = 'Your salary is not enough for children.';
$this->addError('personalSalary', $message);
$this->addError('wifeSalary', $message);
$this->addError('childrenCount', $message);
```

Or use a loop:

```
$attributes = ['personalSalary, 'wifeSalary', 'childrenCount'];
foreach ($attributes as $attribute) {
    $this->addError($attribute, 'Your salary is not enough for children.');
}
```

- Add a common error (not related to particular attribute). We can use the not existing attribute name for adding error, for example `*`, because attribute existence is not checked at that point.

```
$this->addError('*', 'Your salary is not enough for children.');
```

As a result, we will not see error message near form fields. To display it, we can include the error summary in view:

```
<?= $form->errorSummary($model) ?>
```

> Note: Creating validator which validates multiple attributes at once is well described in the community cookbook.

Client-Side Validation

Client-side validation based on JavaScript is desirable when end users provide inputs via HTML forms, because it allows users to find out input errors faster and thus provides a better user experience. You may use or implement a validator that supports client-side validation *in addition to* server-side validation.

> Info: While client-side validation is desirable, it is not a must. Its main purpose is to provide users with a better experience. Similar to input data coming from end users, you should never trust client-side validation. For this reason, you should always perform server-side validation by calling [[yii\base\Model::validate()]], as described in the previous subsections.

Using Client-Side Validation

Many core validators support client-side validation out-of-the-box. All you need to do is just use [[yii\widgets\ActiveForm]] to build your HTML forms. For example, `LoginForm` below declares two rules: one uses the required core validator which is supported on both client and server-sides; the other uses the `validatePassword` inline validator which is only supported on the server side.

```
namespace app\models;

use yii\base\Model;
use app\models\User;

class LoginForm extends Model
{
    public $username;
    public $password;

    public function rules()
    {
        return [
            // username and password are both required
            [['username', 'password'], 'required'],

            // password is validated by validatePassword()
            ['password', 'validatePassword'],
        ];
    }

    public function validatePassword()
    {
        $user = User::findByUsername($this->username);

        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'Incorrect username or password.');
        }
    }
}
```

The HTML form built by the following code contains two input fields `username` and `password`. If you submit the form without entering anything, you will find the error messages requiring you to enter something appear right away without any communication with the server.

```
<?php $form = yii\widgets\ActiveForm::begin(); ?>
    <?= $form->field($model, 'username') ?>
    <?= $form->field($model, 'password')->passwordInput() ?>
    <?= Html::submitButton('Login') ?>
<?php yii\widgets\ActiveForm::end(); ?>
```

Behind the scene, [[yii\widgets\ActiveForm]] will read the validation rules declared in the model and generate appropriate JavaScript code for validators that support client-side validation. When a user changes the value of an input field or submit the form, the client-side validation JavaScript will be triggered.

If you want to turn off client-side validation completely, you may configure the [[yii\widgets\ActiveForm::enableClientValidation]] property to be `false`. You may also turn off client-side validation of individual input fields by configuring their [[yii\widgets\ActiveField::enableClientValidation]] property to be false. When `enableClientValidation` is configured at both the input field level and the form level, the former will take precedence.

> Info: Since version 2.0.11 all validators extending from [[yii\validators\Validator]] receive client-side options from separate method - [[yii\validators\Validator::getClientOptions()]]. You can use it:
>
> - if you want to implement your own custom client-side validation but leave the synchronization with server-side validator options;
> - to extend or customize to fit your specific needs:
>
> ```
> public function getClientOptions($model, $attribute)
> {
>     $options = parent::getClientOptions($model, $attribute);
>     // Modify $options here
>
>     return $options;
> }
> ```

Implementing Client-Side Validation

To create a validator that supports client-side validation, you should implement the [[yii\validators\Validator::clientValidateAttribute()]] method which returns a piece of JavaScript code that performs the validation on the client-side. Within the JavaScript code, you may use the following predefined variables:

- `attribute`: the name of the attribute being validated.
- `value`: the value being validated.
- `messages`: an array used to hold the validation error messages for the attribute.
- `deferred`: an array which deferred objects can be pushed into (explained in the next subsection).

In the following example, we create a `StatusValidator` which validates if an input is a valid status input against the existing status data. The validator supports both server-side and client-side validation.

```
namespace app\components;

use yii\validators\Validator;
use app\models\Status;

class StatusValidator extends Validator
{
    public function init()
    {
        parent::init();
        $this->message = 'Invalid status input.';
    }

    public function validateAttribute($model, $attribute)
    {
        $value = $model->$attribute;
        if (!Status::find()->where(['id' => $value])->exists()) {
            $model->addError($attribute, $this->message);
        }
    }

    public function clientValidateAttribute($model, $attribute, $view)
    {
        $statuses = json_encode(Status::find()->select('id')->asArray()->column());
        $message = json_encode($this->message, JSON_UNESCAPED_SLASHES | JSON_UNESCAPED
_UNICODE);
        return <<<JS
if ($.inArray(value, $statuses) === -1) {
    messages.push($message);
}
JS;
    }
}
```

> Tip: The above code is given mainly to demonstrate how to support client-side validation. In practice, you may use the in core validator to achieve the same goal. You may write the validation rule like the following:
>
> ```
> [
>     ['status', 'in', 'range' => Status::find()->select('id')->asArray()->column()
> ],
> ]
> ```
>
> Tip: If you need to work with client validation manually i.e. dynamically add fields or do some custom UI logic, refer to Working with ActiveForm via JavaScript in Yii 2.0 Cookbook.

Deferred Validation

If you need to perform asynchronous client-side validation, you can create Deferred objects. For example, to perform a custom AJAX validation, you can use the following code:

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.push($.get("/check", {value: value}).done(function(data) {
            if ('' !== data) {
                messages.push(data);
            }
        }));
JS;
}
```

In the above, the `deferred` variable is provided by Yii, which is an array of Deferred objects. The `$.get()` jQuery method creates a Deferred object which is pushed to the `deferred` array.

You can also explicitly create a Deferred object and call its `resolve()` method when the asynchronous callback is hit. The following example shows how to validate the dimensions of an uploaded image file on the client-side.

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        var def = $.Deferred();
        var img = new Image();
        img.onload = function() {
            if (this.width > 150) {
                messages.push('Image too wide!!');
            }
            def.resolve();
        }
        var reader = new FileReader();
        reader.onloadend = function() {
            img.src = reader.result;
        }
        reader.readAsDataURL(file);

        deferred.push(def);
JS;
}
```

> Note: The `resolve()` method must be called after the attribute has been validated. Otherwise the main form validation will not complete.

For simplicity, the `deferred` array is equipped with a shortcut method `add()` which automatically creates a Deferred object and adds it to the `deferred` array. Using this method, you can simplify the above example as follows,

```
public function clientValidateAttribute($model, $attribute, $view)
{
    return <<<JS
        deferred.add(function(def) {
            var img = new Image();
            img.onload = function() {
                if (this.width > 150) {
                    messages.push('Image too wide!!');
                }
                def.resolve();
            }
            var reader = new FileReader();
            reader.onloadend = function() {
                img.src = reader.result;
            }
            reader.readAsDataURL(file);
        });
JS;
}
```

AJAX Validation

Some validations can only be done on the server-side, because only the server has the necessary information. For example, to validate if a username is unique or not, it is necessary to check the user table on the server-side. You can use AJAX-based validation in this case. It will trigger an AJAX request in the background to validate the input while keeping the same user experience as the regular client-side validation.

To enable AJAX validation for a single input field, configure the [[yii\widgets\ActiveField::enableAjaxValidation|enableAjaxValidation]] property of that field to be `true` and specify a unique form `id`:

```
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'registration-form',
]);

echo $form->field($model, 'username', ['enableAjaxValidation' => true]);

// ...

ActiveForm::end();
```

To enable AJAX validation for the whole form, configure [[yii\widgets\ActiveForm::enableAjaxValidation|enableAjaxValidation]] to be `true` at the form level:

```
$form = ActiveForm::begin([
    'id' => 'contact-form',
    'enableAjaxValidation' => true,
]);
```

> Note: When the `enableAjaxValidation` property is configured at both the input field level and the form level, the former will take precedence.

You also need to prepare the server so that it can handle the AJAX validation requests. This can be achieved by a code snippet like the following in the controller actions:

```
if (Yii::$app->request->isAjax && $model->load(Yii::$app->request->post())) {
    Yii::$app->response->format = Response::FORMAT_JSON;
    return ActiveForm::validate($model);
}
```

The above code will check whether the current request is an AJAX. If yes, it will respond to this request by running the validation and returning the errors in JSON format.

> Info: You can also use Deferred Validation to perform AJAX validation. However, the AJAX validation feature described here is more systematic and requires less coding effort.

When both `enableClientValidation` and `enableAjaxValidation` are set to `true`, AJAX validation request will be triggered only after the successful client validation.

Uploading Files

Uploading files in Yii is usually done with the help of [[yii\web\UploadedFile]] which encapsulates each uploaded file as an `UploadedFile` object. Combined with [[yii\widgets\ActiveForm]] and models, you can easily implement a secure file uploading mechanism.

Creating Models

Like working with plain text inputs, to upload a single file you would create a model class and use an attribute of the model to keep the uploaded file instance. You should also declare a validation rule to validate the file upload. For example,

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile
     */
    public $imageFile;

    public function rules()
    {
        return [
            [['imageFile'], 'file', 'skipOnEmpty' => false, 'extensions' => 'png, jpg'
],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            $this->imageFile->saveAs('uploads/' . $this->imageFile->baseName . '.' . $
this->imageFile->extension);
            return true;
        } else {
            return false;
        }
    }
}
```

In the code above, the `imageFile` attribute is used to keep the uploaded file instance. It is associated with a `file` validation rule which uses [[yii\validators\FileValidator]] to ensure a file with extension name `png` or `jpg` is uploaded. The `upload()` method will perform the validation and save the uploaded file on the server.

The `file` validator allows you to check file extensions, size, MIME type, etc. Please refer to the Core Validators section for more details.

> Tip: If you are uploading an image, you may consider using the `image` validator instead. The `image` validator is implemented via [[yii\validators\ImageValidator]] which verifies if an attribute has received a valid image that can be then either saved or processed using the Imagine Extension.

Rendering File Input

Next, create a file input in a view:

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-data']]) ?>


    <?= $form->field($model, 'imageFile')->fileInput() ?>

    <button>Submit</button>

<?php ActiveForm::end() ?>
```

It is important to remember that you add the `enctype` option to the form so that the file can be properly uploaded. The `fileInput()` call will render a `<input type="file">` tag which will allow users to select a file to upload.

> Tip: since version 2.0.8, [[yii\web\widgets\ActiveField::fileInput|fileInput]] adds `enctype` option to the form automatically when file input field is used.

Wiring Up

Now in a controller action, write the code to wire up the model and the view to implement file uploading:

```php
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFile = UploadedFile::getInstance($model, 'imageFile');
            if ($model->upload()) {
                // file is uploaded successfully
                return;
            }
        }

        return $this->render('upload', ['model' => $model]);
    }
}
```

In the above code, when the form is submitted, the [[yii\web\UploadedFile::getInstance()]] method is called to represent the uploaded file as an `UploadedFile` instance. We then rely on the model validation to make sure the uploaded file is valid and save the file on the server.

Uploading Multiple Files

You can also upload multiple files at once, with some adjustments to the code listed in the previous subsections.

First you should adjust the model class by adding the `maxFiles` option in the `file` validation rule to limit the maximum number of files allowed to upload. Setting `maxFiles` to `0` means there is no limit on the number of files that can be uploaded simultaneously. The maximum number of files allowed to be uploaded simultaneously is also limited with PHP directive `max_file_uploads`, which defaults to 20. The `upload()` method should also be updated to save the uploaded files one by one.

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

class UploadForm extends Model
{
    /**
     * @var UploadedFile[]
     */
    public $imageFiles;

    public function rules()
    {
        return [
            [['imageFiles'], 'file', 'skipOnEmpty' => false, 'extensions' => 'png, jpg'
, 'maxFiles' => 4],
        ];
    }

    public function upload()
    {
        if ($this->validate()) {
            foreach ($this->imageFiles as $file) {
                $file->saveAs('uploads/' . $file->baseName . '.' . $file->extension);
            }
            return true;
        } else {
            return false;
        }
    }
}
```

In the view file, you should add the `multiple` option to the `fileInput()` call so that the file upload field can receive multiple files:

```
<?php
use yii\widgets\ActiveForm;
?>

<?php $form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-data']]) ?>


    <?= $form->field($model, 'imageFiles[]')->fileInput(['multiple' => true, 'accept'
=> 'image/*']) ?>

    <button>Submit</button>

<?php ActiveForm::end() ?>
```

And finally in the controller action, you should call `UploadedFile::getInstances()` instead of `UploadedFile::getInstance()` to assign an array of `UploadedFile` instances to `UploadForm::imageFiles`.

```
namespace app\controllers;

use Yii;
use yii\web\Controller;
use app\models\UploadForm;
use yii\web\UploadedFile;

class SiteController extends Controller
{
    public function actionUpload()
    {
        $model = new UploadForm();

        if (Yii::$app->request->isPost) {
            $model->imageFiles = UploadedFile::getInstances($model, 'imageFiles');
            if ($model->upload()) {
                // file is uploaded successfully
                return;
            }
        }

        return $this->render('upload', ['model' => $model]);
    }
}
```

Collecting tabular input

Sometimes you need to handle multiple models of the same kind in a single form. For example, multiple settings, where each setting is stored as a name-value pair and is represented by a `Setting` active record model. This kind of form is also often referred to as "tabular input". In contrast to this, handling different models of different kind, is handled in the section Complex Forms with Multiple Models.

The following shows how to implement tabular input with Yii.

There are three different situations to cover, which have to be handled slightly different:

- Updating a fixed set of records from the database
- Creating a dynamic set of new records
- Updating, creating and deleting of records on one page

In contrast to the single model forms explained before, we are working with an array of models now. This array is passed to the view to display the input fields for each model in a table like style and we will use helper methods of [[yii\base\Model]] that allow loading and validating multiple models at once:

- [[yii\base\Model::loadMultiple()|Model::loadMultiple()]] load post data into an array of models.
- [[yii\base\Model::validateMultiple()|Model::validateMultiple()]] validates an array of models.

Updating a fixed set of records

Let's start with the controller action:

```
<?php

namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use app\models\Setting;

class SettingsController extends Controller
{
    // ...

    public function actionUpdate()
    {
        $settings = Setting::find()->indexBy('id')->all();

        if (Model::loadMultiple($settings, Yii::$app->request->post()) && Model::valid
ateMultiple($settings)) {
            foreach ($settings as $setting) {
                $setting->save(false);
            }
            return $this->redirect('index');
        }

        return $this->render('update', ['settings' => $settings]);
    }
}
```

In the code above we're using [[yii\db\ActiveQuery::indexBy()|indexBy()]] when retrieving models from the database to populate an array indexed by models primary keys. These will be later used to identify form fields. [[yii\base\Model::loadMultiple()|Model::loadMultiple()]] fills multiple models with the form data coming from POST and [[yii\base\Model::validateMultiple()|Model::validateMultiple()]] validates all models at once. As we have validated our models before, using `validateMultiple()`, we're now passing `false` as a parameter to [[yii\db\ActiveRecord::save()|save()]] to not run validation twice.

Now the form that's in `update` view:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin();

foreach ($settings as $index => $setting) {
    echo $form->field($setting, "[$index]value")->label($setting->name);
}

ActiveForm::end();
```

Here for each setting we are rendering name and an input with a value. It is important to add a proper index to input name since that is how [[yii\base\Model::loadMultiple()|Model::loadMultiple()]] determines which model to fill with which values.

Creating a dynamic set of new records

Creating new records is similar to updating, except the part, where we instantiate the models:

```
public function actionCreate()
{
    $count = count(Yii::$app->request->post('Setting', []));
    $settings = [new Setting()];
    for($i = 1; $i < $count; $i++) {
        $settings[] = new Setting();
    }

    // ...
}
```

Here we create an initial `$settings` array containing one model by default so that always at least one text field will be visible in the view. Additionally we add more models for each line of input we may have received.

In the view you can use javascript to add new input lines dynamically.

Combining Update, Create and Delete on one page

> Note: This section is under development.
>
> It has no content yet.

TBD

Getting Data for Multiple Models

When dealing with some complex data, it is possible that you may need to use multiple different models to collect the user input. For example, assuming the user login information is stored in the `user` table while the user profile information is stored in the `profile` table, you may want to collect the input data about a user through a `User` model and a `Profile` model. With the Yii model and form support, you can solve this problem in a way that is not much different from handling a single model.

In the following, we will show how you can create a form that would allow you to collect data for both `User` and `Profile` models.

First, the controller action for collecting the user and profile data can be written as follows,

```
namespace app\controllers;

use Yii;
use yii\base\Model;
use yii\web\Controller;
use yii\web\NotFoundHttpException;
use app\models\User;
use app\models\Profile;

class UserController extends Controller
{
    public function actionUpdate($id)
    {
        $user = User::findOne($id);
        if (!$user) {
            throw new NotFoundHttpException("The user was not found.");
        }

        $profile = Profile::findOne($user->profile_id);

        if (!$profile) {
            throw new NotFoundHttpException("The user has no profile.");
        }

        $user->scenario = 'update';
        $profile->scenario = 'update';

        if ($user->load(Yii::$app->request->post()) && $profile->load(Yii::$app->reque
st->post())) {
            $isValid = $user->validate();
            $isValid = $profile->validate() && $isValid;
            if ($isValid) {
                $user->save(false);
                $profile->save(false);
                return $this->redirect(['user/view', 'id' => $id]);
            }
        }

        return $this->render('update', [
            'user' => $user,
            'profile' => $profile,
        ]);
    }
}
```

In the `update` action, we first load the `$user` and `$profile` models to be updated from the database. We then call [[yii\base\Model::load()]] to populate these two models with the user input. If successful we will validate the two models and save them. Otherwise we will render the `update` view which has the following content:

```
<?php
use yii\helpers\Html;
use yii\widgets\ActiveForm;

$form = ActiveForm::begin([
    'id' => 'user-update-form',
    'options' => ['class' => 'form-horizontal'],
]) ?>
    <?= $form->field($user, 'username') ?>

    ...other input fields...

    <?= $form->field($profile, 'website') ?>

    <?= Html::submitButton('Update', ['class' => 'btn btn-primary']) ?>
<?php ActiveForm::end() ?>
```

As you can see, in the `update` view you would render input fields using two models `$user` and `$profile`.

Data Formatting

To display data in a more readable format for users, you may format them using the `formatter` application component. By default the formatter is implemented by [[yii\i18n\Formatter]] which provides a set of methods to format data as date/time, numbers, currencies, and other commonly used formats. You can use the formatter like the following,

```
$formatter = \Yii::$app->formatter;

// output: January 1, 2014
echo $formatter->asDate('2014-01-01', 'long');

// output: 12.50%
echo $formatter->asPercent(0.125, 2);

// output: <a href="mailto:cebe@example.com">cebe@example.com</a>
echo $formatter->asEmail('cebe@example.com');

// output: Yes
echo $formatter->asBoolean(true);
// it also handles display of null values:

// output: (not set)
echo $formatter->asDate(null);
```

As you can see, all these methods are named as `asXyz()`, where `Xyz` stands for a supported format. Alternatively, you may format data using the generic method [[yii\i18n\Formatter::format()|format()]], which allows you to control the desired format programmatically and is commonly used by widgets like [[yii\grid\GridView]] and [[yii\widgets\DetailView]]. For example,

```
// output: January 1, 2014
echo Yii::$app->formatter->format('2014-01-01', 'date');

// you can also use an array to specify parameters for the format method:
// `2` is the value for the $decimals parameter of the asPercent()-method.
// output: 12.50%
echo Yii::$app->formatter->format(0.125, ['percent', 2]);
```

> Note: The formatter component is designed to format values to be displayed for the end user. If you want to convert user input into machine readable format, or just format a date in a machine readable format, the formatter is not the right tool for that. To convert user input for date and number values you may use [[yii\validators\DateValidator]] and [[yii\validators\NumberValidator]] respectively. For simple conversion between machine readable date and time formats, the PHP date()-function is enough.

Configuring Formatter

You may customize the formatting rules by configuring the `formatter` component in the application configuration. For example,

```
return [
    'components' => [
        'formatter' => [
            'dateFormat' => 'dd.MM.yyyy',
            'decimalSeparator' => ',',
            'thousandSeparator' => ' ',
            'currencyCode' => 'EUR',
        ],
    ],
];
```

Please refer to [[yii\i18n\Formatter]] for the properties that may be configured.

Formatting Date and Time Values

The formatter supports the following output formats that are related with date and time:

- [[yii\i18n\Formatter::asDate()|date]]: the value is formatted as a date, e.g. `January 01, 2014`.
- [[yii\i18n\Formatter::asTime()|time]]: the value is formatted as a time, e.g. `14:23`.
- [[yii\i18n\Formatter::asDatetime()|datetime]]: the value is formatted as date and time, e.g. `January 01, 2014 14:23`.
- [[yii\i18n\Formatter::asTimestamp()|timestamp]]: the value is formatted as a unix timestamp, e.g. `1412609982`.
- [[yii\i18n\Formatter::asRelativeTime()|relativeTime]]: the value is formatted as the time interval between a date and now in human readable form e.g. `1 hour ago`.
- [[yii\i18n\Formatter::asDuration()|duration]]: the value is formatted as a duration in human readable format. e.g. `1 day, 2 minutes`.

The default date and time formats used for the [[yii\i18n\Formatter::asDate()|date]], [[yii\i18n\Formatter::asTime()|time]], and [[yii\i18n\Formatter::asDatetime()|datetime]] methods can be customized globally by configuring [[yii\i18n\Formatter::dateFormat|dateFormat]], [[yii\i18n\Formatter::timeFormat|timeFormat]], and [[yii\i18n\Formatter::datetimeFormat|datetimeFormat]].

You can specify date and time formats using the ICU syntax. You can also use the PHP date() syntax with a prefix `php:` to differentiate it from ICU syntax. For example,

```
// ICU format
echo Yii::$app->formatter->asDate('now', 'yyyy-MM-dd'); // 2014-10-06

// PHP date()-format
echo Yii::$app->formatter->asDate('now', 'php:Y-m-d'); // 2014-10-06
```

When working with applications that need to support multiple languages, you often need to specify different date and time formats for different locales. To simplify this task, you may use format shortcuts (e.g. `long`, `short`), instead. The formatter will turn a format shortcut into an appropriate format according to the currently active [[yii\i18n\Formatter::locale|locale]]. The following format shortcuts are supported (the examples assume `en_GB` is the active locale):

- `short`: will output `06/10/2014` for date and `15:58` for time;
- `medium`: will output `6 Oct 2014` and `15:58:42`;
- `long`: will output `6 October 2014` and `15:58:42 GMT`;
- `full`: will output `Monday, 6 October 2014` and `15:58:42 GMT`.

Since version 2.0.7 it is also possible to format dates in different calendar systems. Please refer to the API documentation of the formatters [[yii\i18n\Formatter::$calendar|$calendar]]-property on how to set a different calendar.

Time Zones

When formatting date and time values, Yii will convert them to the target [[yii\i18n\Formatter::timeZone|time zone]]. The value being formatted is assumed to be in UTC, unless a time zone is explicitly given or you have configured [[yii\i18n\Formatter::defaultTimeZone]].

In the following examples, we assume the target [[yii\i18n\Formatter::timeZone|time zone]] is set as `Europe/Berlin`.

```
// formatting a UNIX timestamp as a time
echo Yii::$app->formatter->asTime(1412599260); // 14:41:00

// formatting a datetime string (in UTC) as a time
echo Yii::$app->formatter->asTime('2014-10-06 12:41:00'); // 14:41:00

// formatting a datetime string (in CEST) as a time
echo Yii::$app->formatter->asTime('2014-10-06 14:41:00 CEST'); // 14:41:00
```

> Note: As time zones are subject to rules made by the governments around the world and may change frequently, it is likely that you do not have the latest information in the time zone database installed on your system. You may refer to the ICU manual for details on updating the time zone database. Please also read Setting up your PHP environment for internationalization.

Formatting Numbers

The formatter supports the following output formats that are related with numbers:

- [[yii\i18n\Formatter::asInteger()|integer]]: the value is formatted as an integer e.g. `42`.
- [[yii\i18n\Formatter::asDecimal()|decimal]]: the value is formatted as a decimal number considering decimal and thousand separators e.g. `2,542.123` or `2.542,123`.
- [[yii\i18n\Formatter::asPercent()|percent]]: the value is formatted as a percent number e.g. `42%`.
- [[yii\i18n\Formatter::asScientific()|scientific]]: the value is formatted as a number in scientific format e.g. `4.2E4`.
- [[yii\i18n\Formatter::asCurrency()|currency]]: the value is formatted as a currency value e.g. `£420.00`. Note that for this function to work properly, the locale needs to include a country part e.g. `en_GB` or `en_US` because language only would be ambiguous in this case.
- [[yii\i18n\Formatter::asSize()|size]]: the value that is a number of bytes is formatted as a human readable size e.g. `410 kibibytes`.
- [[yii\i18n\Formatter::asShortSize()|shortSize]]: is the short version of [[yii\i18n\Formatter::asSize()|size]], e.g. `410 KiB`.

The format for number formatting can be adjusted using the [[yii\i18n\Formatter::decimalSeparator|decimalSeparator]] and [[yii\i18n\Formatter::thousandSeparator|thousandSeparator]], both of which take default values according to the active [[yii\i18n\Formatter::locale|locale]].

For more advanced configuration, [[yii\i18n\Formatter::numberFormatterOptions]] and [[yii\i18n\Formatter::numberFormatterTextOptions]] can be used to configure the NumberFormatter class used internally to implement the formatter. For example, to adjust the maximum and minimum value of fraction digits, you can configure the [[yii\i18n\Formatter::numberFormatterOptions]] property like the following:

```
'numberFormatterOptions' => [
    NumberFormatter::MIN_FRACTION_DIGITS => 0,
    NumberFormatter::MAX_FRACTION_DIGITS => 2,
]
```

Other Formats

Besides date/time and number formats, Yii also supports other commonly used formats, including

- [[yii\i18n\Formatter::asRaw()|raw]]: the value is outputted as is, this is a pseudo-formatter that has no effect except that `null` values will be formatted using [[nullDisplay]].
- [[yii\i18n\Formatter::asText()|text]]: the value is HTML-encoded. This is the default format used by the GridView DataColumn.
- [[yii\i18n\Formatter::asNtext()|ntext]]: the value is formatted as an HTML-encoded plain text with newlines converted into line breaks.
- [[yii\i18n\Formatter::asParagraphs()|paragraphs]]: the value is formatted as HTML-encoded text paragraphs wrapped into `<p>` tags.
- [[yii\i18n\Formatter::asHtml()|html]]: the value is purified using [[HtmlPurifier]] to avoid XSS attacks. You can pass additional options such as `['html', ['Attr.AllowedFrameTargets' => ['_blank']]]`.
- [[yii\i18n\Formatter::asEmail()|email]]: the value is formatted as a `mailto`-link.
- [[yii\i18n\Formatter::asImage()|image]]: the value is formatted as an image tag.
- [[yii\i18n\Formatter::asUrl()|url]]: the value is formatted as a hyperlink.
- [[yii\i18n\Formatter::asBoolean()|boolean]]: the value is formatted as a boolean. By default `true` is rendered as `Yes` and `false` as `No`, translated to the current application language. You can adjust this by configuring the [[yii\i18n\Formatter::booleanFormat]] property.

Null Values

Null values are specially formatted. Instead of displaying an empty string, the formatter will convert it into a preset string which defaults to `(not set)` translated into the current application language. You can configure the [[yii\i18n\Formatter::nullDisplay|nullDisplay]] property to customize this string.

Localizing Data Format

As aforementioned, the formatter may use the currently active [[yii\i18n\Formatter::locale|locale]] to determine how to format a value that is suitable in the target country/region. For example, the same date value may be formatted differently for different locales:

```
Yii::$app->formatter->locale = 'en-US';
echo Yii::$app->formatter->asDate('2014-01-01'); // output: January 1, 2014

Yii::$app->formatter->locale = 'de-DE';
echo Yii::$app->formatter->asDate('2014-01-01'); // output: 1. Januar 2014

Yii::$app->formatter->locale = 'ru-RU';
echo Yii::$app->formatter->asDate('2014-01-01'); // output: 1 января 2014 г.
```

By default, the currently active [[yii\i18n\Formatter::locale|locale]] is determined by the value of [[yii\base\Application::language]]. You may override it by setting the [[yii\i18n\Formatter::locale]] property explicitly.

> Note: The Yii formatter relies on the PHP intl extension to support localized data formatting. Because different versions of the ICU library compiled with PHP may cause different formatting results, it is recommended that you use the same ICU version for all your environments. For more details, please refer to Setting up your PHP environment for internationalization.
>
> If the intl extension is not installed, the data will not be localized.
>
> Note that for date values that are before year 1901 or after 2038, they will not be localized on 32-bit systems, even if the intl extension is installed. This is because in this case ICU is using 32-bit UNIX timestamps to date values.

Pagination

When there are too much data to be displayed on a single page, a common strategy is to display them in multiple pages and on each page only display a small portion of the data. This strategy is known as *pagination*.

Yii uses a [[yii\data\Pagination]] object to represent the information about a pagination scheme. In particular,

- [[yii\data\Pagination::$totalCount|total count]] specifies the total number of data items. Note that this is usually much more than the number of data items needed to display on a single page.
- [[yii\data\Pagination::$pageSize|page size]] specifies how many data items each page contains. The default value is 20.
- [[yii\data\Pagination::$page|current page]] gives the current page number (zero-based). The default value is 0, meaning the first page.

With a fully specified [[yii\data\Pagination]] object, you can retrieve and display data partially. For example, if you are fetching data from a database, you can specify the `OFFSET` and `LIMIT` clause of the DB query with the corresponding values provided by the pagination. Below is an example,

```
use yii\data\Pagination;

// build a DB query to get all articles with status = 1
$query = Article::find()->where(['status' => 1]);

// get the total number of articles (but do not fetch the article data yet)
$count = $query->count();

// create a pagination object with the total count
$pagination = new Pagination(['totalCount' => $count]);

// limit the query using the pagination and retrieve the articles
$articles = $query->offset($pagination->offset)
    ->limit($pagination->limit)
    ->all();
```

Which page of articles will be returned in the above example? It depends on whether a query parameter named `page` is given. By default, the pagination will attempt to set the [[yii\data\Pagination::$page|current page]] to be the value of the `page` parameter. If the parameter is not provided, then it will default to 0.

To facilitate building the UI element that supports pagination, Yii provides the [[yii\widgets\LinkPager]] widget that displays a list of page buttons upon which users can click to indicate which page of data should be displayed. The widget takes a pagination object so that it knows what is the current page and how many page buttons should be displayed. For example,

```
use yii\widgets\LinkPager;

echo LinkPager::widget([
    'pagination' => $pagination,
]);
```

If you want to build UI element manually, you may use [[yii\data\Pagination::createUrl()]] to create URLs that would lead to different pages. The method requires a page parameter and will create a properly formatted URL containing the page parameter. For example,

```
// specifies the route that the URL to be created should use
// If you do not specify this, the currently requested route will be used
$pagination->route = 'article/index';

// displays: /index.php?r=article%2Findex&page=100
echo $pagination->createUrl(100);

// displays: /index.php?r=article%2Findex&page=101
echo $pagination->createUrl(101);
```

> Tip: You can customize the name of the `page` query parameter by configuring the [[yii\data\Pagination::pageParam|pageParam]] property when creating the pagination object.

Sorting

When displaying multiple rows of data, it is often needed that the data be sorted according to some columns specified by end users. Yii uses a [[yii\data\Sort]] object to represent the information about a sorting schema. In particular,

- [[yii\data\Sort::$attributes|attributes]] specifies the *attributes* by which the data can be sorted. An attribute can be as simple as a model attribute. It can also be a composite one by combining multiple model attributes or DB columns. More details will be given in the following.
- [[yii\data\Sort::$attributeOrders|attributeOrders]] gives the currently requested ordering directions for each attribute.
- [[yii\data\Sort::$orders|orders]] gives the ordering directions in terms of the low-level columns.

To use [[yii\data\Sort]], first declare which attributes can be sorted. Then retrieve the currently requested ordering information from [[yii\data\Sort::$attributeOrders|attributeOrders]] or [[yii\data\Sort::$orders|orders]] and use them to customize the data query. For example,

```
use yii\data\Sort;

$sort = new Sort([
    'attributes' => [
        'age',
        'name' => [
            'asc' => ['first_name' => SORT_ASC, 'last_name' => SORT_ASC],
            'desc' => ['first_name' => SORT_DESC, 'last_name' => SORT_DESC],
            'default' => SORT_DESC,
            'label' => 'Name',
        ],
    ],
]);

$articles = Article::find()
    ->where(['status' => 1])
    ->orderBy($sort->orders)
    ->all();
```

In the above example, two attributes are declared for the [[yii\data\Sort|Sort]] object: `age` and `name`.

The `age` attribute is a *simple* attribute corresponding to the `age` attribute of the `Article` Active Record class. It is equivalent to the following declaration:

```
'age' => [
    'asc' => ['age' => SORT_ASC],
    'desc' => ['age' => SORT_DESC],
    'default' => SORT_ASC,
    'label' => Inflector::camel2words('age'),
]
```

The `name` attribute is a *composite* attribute defined by `first_name` and `last_name` of `Article`. It is declared using the following array structure:

- The `asc` and `desc` elements specify how to sort by the attribute in ascending and descending directions, respectively. Their values represent the actual columns and the directions by which the data should be sorted by. You can specify one or multiple columns to indicate simple ordering or composite ordering.
- The `default` element specifies the direction by which the attribute should be sorted when initially requested. It defaults to ascending order, meaning if it is not sorted before and you request to sort by this attribute, the data will be sorted by this attribute in ascending order.
- The `label` element specifies what label should be used when calling [[yii\data\Sort::link()]] to create a sort link. If not set, [[yii\helpers\Inflector::camel2words()]] will be called to generate a label from the attribute name. Note that it will not be HTML-encoded.

> Info: You can directly feed the value of [[yii\data\Sort::$orders|orders]] to the database query to build its `ORDER BY` clause. Do not use [[yii\data\Sort::$attributeOrders|attributeOrders]] because some of the attributes may be composite and cannot be recognized by the database query.

You can call [[yii\data\Sort::link()]] to generate a hyperlink upon which end users can click to request sorting the data by the specified attribute. You may also call [[yii\data\Sort::createUrl()]] to create a sortable URL. For example,

```
// specifies the route that the URL to be created should use
// If you do not specify this, the currently requested route will be used
$sort->route = 'article/index';

// display links leading to sort by name and age, respectively
echo $sort->link('name') . ' | ' . $sort->link('age');

// displays: /index.php?r=article%2Findex&sort=age
echo $sort->createUrl('age');
```

[[yii\data\Sort]] checks the `sort` query parameter to determine which attributes are being requested for sorting. You may specify a default ordering via [[yii\data\Sort::defaultOrder]] when the query parameter is not present. You may also customize the name of the query parameter by configuring the [[yii\data\Sort::sortParam|sortParam]] property.

Data Providers

In the Pagination and Sorting sections, we have described how to allow end users to choose a particular page of data to display and sort them by some columns. Because the task of paginating and sorting data is very common, Yii provides a set of *data provider* classes to encapsulate it.

A data provider is a class implementing [[yii\data\DataProviderInterface]]. It mainly supports retrieving paginated and sorted data. It is usually used to work with data widgets so that end users can interactively paginate and sort data.

The following data provider classes are included in the Yii releases:

- [[yii\data\ActiveDataProvider]]: uses [[yii\db\Query]] or [[yii\db\ActiveQuery]] to query data from databases and return them in terms of arrays or Active Record instances.
- [[yii\data\SqlDataProvider]]: executes a SQL statement and returns database data as arrays.
- [[yii\data\ArrayDataProvider]]: takes a big array and returns a slice of it based on the paginating and sorting specifications.

The usage of all these data providers share the following common pattern:

```
// create the data provider by configuring its pagination and sort properties
$provider = new XyzDataProvider([
    'pagination' => [...],
    'sort' => [...],
]);

// retrieves paginated and sorted data
$models = $provider->getModels();

// get the number of data items in the current page
$count = $provider->getCount();

// get the total number of data items across all pages
$totalCount = $provider->getTotalCount();
```

You specify the pagination and sorting behaviors of a data provider by configuring its [[yii\data\BaseDataProvider::pagination|pagination]] and [[yii\data\BaseDataProvider::sort|sort]] properties which correspond to the configurations for [[yii\data\Pagination]] and [[yii\data\Sort]], respectively. You may also configure them to be `false` to disable pagination and/or sorting features.

Data widgets, such as [[yii\grid\GridView]], have a property named `dataProvider` which can take a data provider instance and display the data it provides. For example,

```
echo yii\grid\GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

These data providers mainly vary in the way how the data source is specified. In the following subsections, we will explain the detailed usage of each of these data providers.

Active Data Provider

To use [[yii\data\ActiveDataProvider]], you should configure its [[yii\data\ActiveDataProvider::query|query]] property. It can take either a [[yii\db\Query]] or [[yii\db\ActiveQuery]] object. If the former, the data returned will be arrays; if the latter, the data returned can be either arrays or Active Record instances. For example,

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'defaultOrder' => [
            'created_at' => SORT_DESC,
            'title' => SORT_ASC,
        ]
    ],
]);

// returns an array of Post objects
$posts = $provider->getModels();
```

If `$query` in the above example is created using the following code, then the data provider will return raw arrays.

```
use yii\db\Query;

$query = (new Query())->from('post')->where(['status' => 1]);
```

> Note: If a query already specifies the `orderBy` clause, the new ordering instructions given by end users (through the `sort` configuration) will be appended to the existing `orderBy` clause. Any existing `limit` and `offset` clauses will be overwritten by the pagination request from end users (through the `pagination` configuration).

By default, [[yii\data\ActiveDataProvider]] uses the `db` application component as the database connection. You may use a different database connection by configuring the [[yii\data\ActiveDataProvider::db]] property.

SQL Data Provider

[[yii\data\SqlDataProvider]] works with a raw SQL statement which is used to fetch the needed data. Based on the specifications of [[yii\data\SqlDataProvider::sort|sort]] and [[yii\data\SqlDataProvider::pagination|pagination]], the provider will adjust the `ORDER BY` and `LIMIT` clauses of the SQL statement accordingly to fetch only the requested page of data in the desired order.

To use [[yii\data\SqlDataProvider]], you should specify the [[yii\data\SqlDataProvider::sql|sql]] property as well as the [[yii\data\SqlDataProvider::totalCount|totalCount]] property. For example,

```
use yii\data\SqlDataProvider;

$count = Yii::$app->db->createCommand('
    SELECT COUNT(*) FROM post WHERE status=:status
', [':status' => 1])->queryScalar();

$provider = new SqlDataProvider([
    'sql' => 'SELECT * FROM post WHERE status=:status',
    'params' => [':status' => 1],
    'totalCount' => $count,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => [
            'title',
            'view_count',
            'created_at',
        ],
    ],
]);

// returns an array of data rows
$models = $provider->getModels();
```

> Info: The [[yii\data\SqlDataProvider::totalCount|totalCount]] property is required only if you need to paginate the data. This is because the SQL statement specified via [[yii\data\SqlDataProvider::sql|sql]] will be modified by the provider to return only the currently requested page of data. The provider still needs to know the total number of data items in order to correctly calculate the number of pages available.

Array Data Provider

[[yii\data\ArrayDataProvider]] is best used when working with a big array. The provider allows you to return a page of the array data sorted by one or multiple columns. To use [[yii\data\ArrayDataProvider]], you should specify the [[yii\data\ArrayDataProvider::allModels|allModels]] property as the big array. Elements in the big array can be either associative arrays (e.g. query results of DAO) or objects (e.g. Active Record instances). For example,

```
use yii\data\ArrayDataProvider;

$data = [
    ['id' => 1, 'name' => 'name 1', ...],
    ['id' => 2, 'name' => 'name 2', ...],
    ...
    ['id' => 100, 'name' => 'name 100', ...],
];

$provider = new ArrayDataProvider([
    'allModels' => $data,
    'pagination' => [
        'pageSize' => 10,
    ],
    'sort' => [
        'attributes' => ['id', 'name'],
    ],
]);

// get the rows in the currently requested page
$rows = $provider->getModels();
```

> Note: Compared to Active Data Provider and SQL Data Provider, array data provider is less efficient because it requires loading *all* data into the memory.

Working with Data Keys

When using the data items returned by a data provider, you often need to identify each data item with a unique key. For example, if the data items represent customer information, you may want to use the customer ID as the key for each customer data. Data providers can return a list of such keys corresponding with the data items returned by [[yii\data\DataProviderInterface::getModels()]]. For example,

```
use yii\data\ActiveDataProvider;

$query = Post::find()->where(['status' => 1]);

$provider = new ActiveDataProvider([
    'query' => $query,
]);

// returns an array of Post objects
$posts = $provider->getModels();

// returns the primary key values corresponding to $posts
$ids = $provider->getKeys();
```

In the above example, because you provide to [[yii\data\ActiveDataProvider]] an [[yii\db\ActiveQuery]] object, it is intelligent enough to return primary key values as the keys. You may also explicitly specify how the key values should be calculated by configuring [[yii\data\ActiveDataProvider::key]] with a column name or a callable calculating key values. For example,

```
// use "slug" column as key values
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => 'slug',
]);

// use the result of md5(id) as key values
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'key' => function ($model) {
        return md5($model->id);
    }
]);
```

Creating Custom Data Provider

To create your own custom data provider classes, you should implement [[yii\data\DataProviderInterface]]. An easier way is to extend from [[yii\data\BaseDataProvider]] which allows you to focus on the core data provider logic. In

particular, you mainly need to implement the following methods:

- [[yii\data\BaseDataProvider::prepareModels()|prepareModels()]]: prepares the data models that will be made available in the current page and returns them as an array.
- [[yii\data\BaseDataProvider::prepareKeys()|prepareKeys()]]: accepts an array of currently available data models and returns keys associated with them.
- [[yii\data\BaseDataProvider::prepareTotalCount()|prepareTotalCount]]: returns a value indicating the total number of data models in the data provider.

Below is an example of a data provider that reads CSV data efficiently:

```
<?php
use yii\data\BaseDataProvider;

class CsvDataProvider extends BaseDataProvider
{
    /**
     * @var string name of the CSV file to read
     */
    public $filename;

    /**
     * @var string|callable name of the key column or a callable returning it
     */
    public $key;

    /**
     * @var SplFileObject
     */
    protected $fileObject; // SplFileObject is very convenient for seeking to particul
ar line in a file


    /**
     * @inheritdoc
     */
    public function init()
    {
        parent::init();

        // open file
        $this->fileObject = new SplFileObject($this->filename);
    }

    /**
     * @inheritdoc
     */
    protected function prepareModels()
    {
        $models = [];
        $pagination = $this->getPagination();
```

```
        if ($pagination === false) {
            // in case there's no pagination, read all lines
            while (!$this->fileObject->eof()) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        } else {
            // in case there's pagination, read only a single page
            $pagination->totalCount = $this->getTotalCount();
            $this->fileObject->seek($pagination->getOffset());
            $limit = $pagination->getLimit();

            for ($count = 0; $count < $limit; ++$count) {
                $models[] = $this->fileObject->fgetcsv();
                $this->fileObject->next();
            }
        }

        return $models;
    }

    /**
     * @inheritdoc
     */
    protected function prepareKeys($models)
    {
        if ($this->key !== null) {
            $keys = [];

            foreach ($models as $model) {
                if (is_string($this->key)) {
                    $keys[] = $model[$this->key];
                } else {
                    $keys[] = call_user_func($this->key, $model);
                }
            }

            return $keys;
        } else {
            return array_keys($models);
        }
    }

    /**
     * @inheritdoc
     */
    protected function prepareTotalCount()
    {
        $count = 0;

        while (!$this->fileObject->eof()) {
            $this->fileObject->next();
```

```
            ++$count;
        }

        return $count;
    }
}
```

Data widgets

Yii provides a set of widgets that can be used to display data. While the DetailView widget can be used to display data for a single record, ListView and GridView can be used to display a list or table of data records providing features like pagination, sorting and filtering.

DetailView

The [[yii\widgets\DetailView|DetailView]] widget displays the details of a single data [[yii\widgets\DetailView::$model|model]].

It is best used for displaying a model in a regular format (e.g. each model attribute is displayed as a row in a table). The model can be either an instance or subclass of [[\yii\base\Model]] such as an active record or an associative array.

DetailView uses the [[yii\widgets\DetailView::$attributes|$attributes]] property to determine which model attributes should be displayed and how they should be formatted. See the formatter section for available formatting options.

A typical usage of DetailView is as follows:

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        'title',                                        // title attribute (in plai
n text)
        'description:html',                             // description attribute fo
rmatted as HTML
        [                                               // the owner name of the mo
del
            'label' => 'Owner',
            'value' => $model->owner->name,
            'contentOptions' => ['class' => 'bg-red'],     // HTML attributes to custo
mize value tag
            'captionOptions' => ['tooltip' => 'Tooltip'],  // HTML attributes to custo
mize label tag
        ],
        'created_at:datetime',                          // creation date formatted
as datetime
    ],
]);
```

Remember that unlike [[yii\widgets\GridView|GridView]] which processes a set of models, [[yii\widgets\DetailView|DetailView]] processes just one. So most of the time there is no need for using closure since `$model` is the only one model for display and available in view as a variable.

However some cases can make using of closure useful. For example when `visible` is specified and you want to prevent `value` calculations in case it evaluates to `false`:

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        [
            'attribute' => 'owner',
            'value' => function ($model) {
                return $model->owner->name;
            },
            'visible' => \Yii::$app->user->can('posts.owner.view'),
        ],
    ],
]);
```

ListView

The [[yii\widgets\ListView|ListView]] widget is used to display data from a data provider. Each data model is rendered using the specified [[yii\widgets\ListView::$itemView|view file]]. Since it provides features such as pagination, sorting and filtering out of the box, it is handy both to display information to end user and to create data managing UI.

A typical usage is as follows:

```
use yii\widgets\ListView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
]);
```

The `_post` view file could contain the following:

```
<?php
use yii\helpers\Html;
use yii\helpers\HtmlPurifier;
?>
<div class="post">
    <h2><?= Html::encode($model->title) ?></h2>

    <?= HtmlPurifier::process($model->text) ?>
</div>
```

In the view file above, the current data model is available as `$model`. Additionally the following variables are available:

- `$key`: mixed, the key value associated with the data item.
- `$index`: integer, the zero-based index of the data item in the items array returned by the data provider.
- `$widget`: ListView, this widget instance.

If you need to pass additional data to each view, you can use the [[yii\widgets\ListView::$viewParams|$viewParams]] property to pass key value pairs like the following:

```
echo ListView::widget([
    'dataProvider' => $dataProvider,
    'itemView' => '_post',
    'viewParams' => [
        'fullView' => true,
        'context' => 'main-page',
        // ...
    ],
]);
```

These are then also available as variables in the view.

GridView

Data grid or [[yii\grid\GridView|GridView]] is one of the most powerful Yii widgets. It is extremely useful if you need to quickly build the admin section of the system. It takes data from a data provider and renders each row using a set of [[yii\grid\GridView::columns|columns]] presenting data in the form of a table.

Each row of the table represents the data of a single data item, and a column usually represents an attribute of the item (some columns may correspond to complex expressions of attributes or static text).

The minimal code needed to use GridView is as follows:

```
use yii\grid\GridView;
use yii\data\ActiveDataProvider;

$dataProvider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);
echo GridView::widget([
    'dataProvider' => $dataProvider,
]);
```

The above code first creates a data provider and then uses GridView to display every attribute in every row taken from the data provider. The displayed table is equipped with sorting and pagination functionality out of the box.

Grid columns

The columns of the grid table are configured in terms of [[yii\grid\Column]] classes, which are configured in the [[yii\grid\GridView::columns|columns]] property of GridView configuration. Depending on column type and settings these are able to present data differently. The default class is [[yii\grid\DataColumn]], which represents a model attribute and can be sorted and filtered by.

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // Simple columns defined by the data contained in $dataProvider.
        // Data from the model's column will be used.
        'id',
        'username',
        // More complex one.
        [
            'class' => 'yii\grid\DataColumn', // can be omitted, as it is the default
            'value' => function ($data) {
                return $data->name; // $data['name'] for array data, e.g. using SqlDat
aProvider.
            },
        ],
    ],
]);
```

Note that if the [[yii\grid\GridView::columns|columns]] part of the configuration isn't specified, Yii tries to show all possible columns of the data provider's model.

Column classes

Grid columns could be customized by using different column classes:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\SerialColumn', // <-- here
            // you may configure additional properties here
        ],
```

In addition to column classes provided by Yii that we'll review below, you can create your own column classes.

Each column class extends from [[yii\grid\Column]] so that there are some common options you can set while configuring grid columns.

- [[yii\grid\Column::header|header]] allows to set content for header row.
- [[yii\grid\Column::footer|footer]] allows to set content for footer row.
- [[yii\grid\Column::visible|visible]] defines if the column should be visible.
- [[yii\grid\Column::content|content]] allows you to pass a valid PHP callback that will return data for a row. The format is the following:

  ```
  function ($model, $key, $index, $column) {
      return 'a string';
  }
  ```

You may specify various container HTML options by passing arrays to:

- [[yii\grid\Column::headerOptions|headerOptions]]
- [[yii\grid\Column::footerOptions|footerOptions]]
- [[yii\grid\Column::filterOptions|filterOptions]]
- [[yii\grid\Column::contentOptions|contentOptions]]

Data column

[[yii\grid\DataColumn|Data column]] is used for displaying and sorting data. It is the default column type so the specifying class could be omitted when using it.

The main setting of the data column is its [[yii\grid\DataColumn::format|format]] property. Its values correspond to methods in the `formatter` application component that is [[\yii\i18n\Formatter|Formatter]] by default:

```
echo GridView::widget([
    'columns' => [
        [
            'attribute' => 'name',
            'format' => 'text'
        ],
        [
            'attribute' => 'birthday',
            'format' => ['date', 'php:Y-m-d']
        ],
    ],
]);
```

In the above, `text` corresponds to [[\yii\i18n\Formatter::asText()]]. The value of the column is passed as the first argument. In the second column definition, `date` corresponds to [[\yii\i18n\Formatter::asDate()]]. The value of the column is, again, passed as the first argument while 'php:Y-m-d' is used as the second argument value.

For a list of available formatters see the section about Data Formatting.

For configuring data columns there is also a shortcut format which is described in the API documentation for [[yii\grid\GridView::columns|columns]].

Action column

[[yii\grid\ActionColumn|Action column]] displays action buttons such as update or delete for each row.

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        [
            'class' => 'yii\grid\ActionColumn',
            // you may configure additional properties here
        ],
```

Available properties you can configure are:

- [[yii\grid\ActionColumn::controller|controller]] is the ID of the controller that should handle the actions. If not set, it will use the currently active controller.
- [[yii\grid\ActionColumn::template|template]] defines the template used for composing

each cell in the action column. Tokens enclosed within curly brackets are treated as controller action IDs (also called *button names* in the context of action column). They will be replaced by the corresponding button rendering callbacks specified in [[yii\grid\ActionColumn::$buttons|buttons]]. For example, the token `{view}` will be replaced by the result of the callback `buttons['view']`. If a callback cannot be found, the token will be replaced with an empty string. The default tokens are `{view} {update} {delete}`.

- [[yii\grid\ActionColumn::buttons|buttons]] is an array of button rendering callbacks. The array keys are the button names (without curly brackets), and the values are the corresponding button rendering callbacks. The callbacks should use the following signature:

```
function ($url, $model, $key) {
    // return the button HTML code
}
```

In the code above, `$url` is the URL that the column creates for the button, `$model` is the model object being rendered for the current row, and `$key` is the key of the model in the data provider array.

- [[yii\grid\ActionColumn::urlCreator|urlCreator]] is a callback that creates a button URL using the specified model information. The signature of the callback should be the same as that of [[yii\grid\ActionColumn::createUrl()]]. If this property is not set, button URLs will be created using [[yii\grid\ActionColumn::createUrl()]].

- [[yii\grid\ActionColumn::visibleButtons|visibleButtons]] is an array of visibility conditions for each button. The array keys are the button names (without curly brackets), and the values are the boolean `true` / `false` or the anonymous function. When the button name is not specified in this array it will be shown by default. The callbacks must use the following signature:

```
function ($model, $key, $index) {
    return $model->status === 'editable';
}
```

Or you can pass a boolean value:

```
[
    'update' => \Yii::$app->user->can('update')
]
```

Checkbox column

[[yii\grid\CheckboxColumn|Checkbox column]] displays a column of checkboxes.

To add a CheckboxColumn to the GridView, add it to the [[yii\grid\GridView::$columns|columns]] configuration as follows:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        // ...
        [
            'class' => 'yii\grid\CheckboxColumn',
            // you may configure additional properties here
        ],
    ],
```

Users may click on the checkboxes to select rows of the grid. The selected rows may be obtained by calling the following JavaScript code:

```
var keys = $('#grid').yiiGridView('getSelectedRows');
// keys is an array consisting of the keys associated with the selected rows
```

Serial column

[[yii\grid\SerialColumn|Serial column]] renders row numbers starting with `1` and going forward.

Usage is as simple as the following:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'], // <-- here
        // ...
```

Sorting data

> Note: This section is under development.
>
> - https://github.com/yiisoft/yii2/issues/1576

Filtering data

For filtering data, the GridView needs a model that represents the search criteria which is usually taken from the filter fields in the GridView table. A common practice when using active records is to create a search Model class that provides needed functionality (it can be generated for you by Gii). This class defines the validation rules to show filter controls on the GridView table and to provide a `search()` method that will return the data provider with an adjusted query that processes the search criteria.

To add the search capability for the `Post` model, we can create a `PostSearch` model like the following example:

```php
<?php

namespace app\models;

use Yii;
use yii\base\Model;
use yii\data\ActiveDataProvider;

class PostSearch extends Post
{
    public function rules()
    {
        // only fields in rules() are searchable
        return [
            [['id'], 'integer'],
            [['title', 'creation_date'], 'safe'],
        ];
    }

    public function scenarios()
    {
        // bypass scenarios() implementation in the parent class
        return Model::scenarios();
    }

    public function search($params)
    {
        $query = Post::find();

        $dataProvider = new ActiveDataProvider([
            'query' => $query,
        ]);

        // load the search form data and validate
        if (!($this->load($params) && $this->validate())) {
            return $dataProvider;
        }

        // adjust the query by adding the filters
        $query->andFilterWhere(['id' => $this->id]);
        $query->andFilterWhere(['like', 'title', $this->title])
              ->andFilterWhere(['like', 'creation_date', $this->creation_date]);

        return $dataProvider;
    }
}
```

> Tip: See Query Builder and especially Filter Conditions to learn how to build filtering query.

You can use this function in the controller to get the dataProvider for the GridView:

```
$searchModel = new PostSearch();
$dataProvider = $searchModel->search(Yii::$app->request->get());

return $this->render('myview', [
    'dataProvider' => $dataProvider,
    'searchModel' => $searchModel,
]);
```

And in the view you then assign the `$dataProvider` and `$searchModel` to the GridView:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'filterModel' => $searchModel,
    'columns' => [
        // ...
    ],
]);
```

Separate filter form

Most of the time using GridView header filters is enough, but in case you need a separate filter form, you can easily add it as well. You can create partial view `_search.php` with the following contents:

```
<?php

use yii\helpers\Html;
use yii\widgets\ActiveForm;

/* @var $this yii\web\View */
/* @var $model app\models\PostSearch */
/* @var $form yii\widgets\ActiveForm */
?>

<div class="post-search">
    <?php $form = ActiveForm::begin([
        'action' => ['index'],
        'method' => 'get',
    ]); ?>

    <?= $form->field($model, 'title') ?>

    <?= $form->field($model, 'creation_date') ?>

    <div class="form-group">
        <?= Html::submitButton('Search', ['class' => 'btn btn-primary']) ?>
        <?= Html::submitButton('Reset', ['class' => 'btn btn-default']) ?>
    </div>

    <?php ActiveForm::end(); ?>
</div>
```

and include it in `index.php` view like so:

```
<?= $this->render('_search', ['model' => $searchModel]) ?>
```

> Note: if you use Gii to generate CRUD code, the separate filter form (`_search.php`) is generated by default, but is commented in `index.php` view. Uncomment it and it's ready to use!

Separate filter form is useful when you need to filter by fields, that are not displayed in GridView or for special filtering conditions, like date range. For filtering by date range we can add non DB attributes `createdFrom` and `createdTo` to the search model:

```
class PostSearch extends Post
{
    /**
     * @var string
     */
    public $createdFrom;

    /**
     * @var string
     */
    public $createdTo;
}
```

Extend query conditions in the `search()` method like so:

```
$query->andFilterWhere(['>=', 'creation_date', $this->createdFrom])
      ->andFilterWhere(['<=', 'creation_date', $this->createdTo]);
```

And add the representative fields to the filter form:

```
<?= $form->field($model, 'creationFrom') ?>

<?= $form->field($model, 'creationTo') ?>
```

Working with model relations

When displaying active records in a GridView you might encounter the case where you display values of related columns such as the post author's name instead of just his `id`. You do this by defining the attribute name in [[yii\grid\GridView::$columns]] as `author.name` when the `Post` model has a relation named `author` and the author model has an attribute `name`. The GridView will then display the name of the author but sorting and filtering are not enabled by default. You have to adjust the `PostSearch` model that has been introduced in the last section to add this functionality.

To enable sorting on a related column you have to join the related table and add the sorting rule to the Sort component of the data provider:

```
$query = Post::find();
$dataProvider = new ActiveDataProvider([
    'query' => $query,
]);

// join with relation `author` that is a relation to the table `users`
// and set the table alias to be `author`
$query->joinWith(['author' => function($query) { $query->from(['author' => 'users']);
}]);
// since version 2.0.7, the above line can be simplified to $query->joinWith('author A
S author');
// enable sorting for the related column
$dataProvider->sort->attributes['author.name'] = [
    'asc' => ['author.name' => SORT_ASC],
    'desc' => ['author.name' => SORT_DESC],
];

// ...
```

Filtering also needs the joinWith call as above. You also need to define the searchable column in attributes and rules like this:

```
public function attributes()
{
    // add related fields to searchable attributes
    return array_merge(parent::attributes(), ['author.name']);
}

public function rules()
{
    return [
        [['id'], 'integer'],
        [['title', 'creation_date', 'author.name'], 'safe'],
    ];
}
```

In `search()` you then just add another filter condition with:

```
$query->andFilterWhere(['LIKE', 'author.name', $this->getAttribute('author.name')]);
```

> Info: In the above we use the same string for the relation name and the table alias; however, when your alias and relation name differ, you have to pay attention to where you use the alias and where you use the relation name. A simple rule for this is to use the alias in every place that is used to build the database query and the relation name in all other definitions such as `attributes()` and `rules()` etc.
>
> For example, if you use the alias `au` for the author relation table, the joinWith statement looks like the following:
>
> ```
> $query->joinWith(['author au']);
> ```
>
> It is also possible to just call `$query->joinWith(['author']);` when the alias is defined in the relation definition.
>
> The alias has to be used in the filter condition but the attribute name stays the same:
>
> ```
> $query->andFilterWhere(['LIKE', 'au.name', $this->getAttribute('author.name')]);
> ```
>
> The same is true for the sorting definition:
>
> ```
> $dataProvider->sort->attributes['author.name'] = [
>     'asc' => ['au.name' => SORT_ASC],
>     'desc' => ['au.name' => SORT_DESC],
> ];
> ```
>
> Also, when specifying the [[yii\data\Sort::defaultOrder|defaultOrder]] for sorting, you need to use the relation name instead of the alias:
>
> ```
> $dataProvider->sort->defaultOrder = ['author.name' => SORT_ASC];
> ```
>
> Info: For more information on `joinWith` and the queries performed in the background, check the active record docs on joining with relations.

Using SQL views for filtering, sorting and displaying data

There is also another approach that can be faster and more useful - SQL views. For example, if we need to show the gridview with users and their profiles, we can do so in this way:

```
CREATE OR REPLACE VIEW vw_user_info AS
    SELECT user.*, user_profile.lastname, user_profile.firstname
    FROM user, user_profile
    WHERE user.id = user_profile.user_id
```

Then you need to create the ActiveRecord that will be representing this view:

```
namespace app\models\views\grid;

use yii\db\ActiveRecord;

class UserView extends ActiveRecord
{

    /**
     * @inheritdoc
     */
    public static function tableName()
    {
        return 'vw_user_info';
    }

    public static function primaryKey()
    {
        return ['id'];
    }

    /**
     * @inheritdoc
     */
    public function rules()
    {
        return [
            // define here your rules
        ];
    }

    /**
     * @inheritdoc
     */
    public static function attributeLabels()
    {
        return [
            // define here your attribute labels
        ];
    }


}
```

After that you can use this UserView active record with search models, without additional specification of sorting and filtering attributes. All attributes will be working out of the box. Note that this approach has several pros and cons:

- you don't need to specify different sorting and filtering conditions. Everything works out of the box;
- it can be much faster because of the data size, count of sql queries performed (for each relation you will not need any additional query);
- since this is just a simple mapping UI on the sql view it lacks some domain logic that is in your entities, so if you have some methods like `isActive`, `isDeleted` or others that will influence the UI, you will need to duplicate them in this class too.

Multiple GridViews on one page

You can use more than one GridView on a single page but some additional configuration is needed so that they do not interfere with each other. When using multiple instances of GridView you have to configure different parameter names for the generated sort and pagination links so that each GridView has its own individual sorting and pagination. You do so by setting the [[yii\data\Sort::sortParam|sortParam]] and [[yii\data\Pagination::pageParam|pageParam]] of the dataProvider's [[yii\data\BaseDataProvider::$sort|sort]] and [[yii\data\BaseDataProvider::$pagination|pagination]] instances.

Assume we want to list the `Post` and `User` models for which we have already prepared two data providers in `$userProvider` and `$postProvider`:

```
use yii\grid\GridView;

$userProvider->pagination->pageParam = 'user-page';
$userProvider->sort->sortParam = 'user-sort';

$postProvider->pagination->pageParam = 'post-page';
$postProvider->sort->sortParam = 'post-sort';

echo '<h1>Users</h1>';
echo GridView::widget([
    'dataProvider' => $userProvider,
]);

echo '<h1>Posts</h1>';
echo GridView::widget([
    'dataProvider' => $postProvider,
]);
```

Using GridView with Pjax

The [[yii\widgets\Pjax|Pjax]] widget allows you to update a certain section of a page instead of reloading the entire page. You can use it to update only the [[yii\grid\GridView|GridView]] content when using filters.

```
use yii\widgets\Pjax;
use yii\grid\GridView;

Pjax::begin([
    // PJax options
]);
    Gridview::widget([
        // GridView options
    ]);
Pjax::end();
```

Pjax also works for the links inside the [[yii\widgets\Pjax|Pjax]] widget and for the links specified by [[yii\widgets\Pjax::$linkSelector|Pjax::$linkSelector]]. But this might be a problem for the links of an [[yii\grid\ActionColumn|ActionColumn]]. To prevent this, add the HTML attribute `data-pjax="0"` to the links when you edit the [[yii\grid\ActionColumn::$buttons|ActionColumn::$buttons]] property.

GridView/ListView with Pjax in Gii

Since 2.0.5, the CRUD generator of Gii has an option called `$enablePjax` that can be used via either web interface or command line.

```
yii gii/crud --controllerClass="backend\\controllers\PostController" \
  --modelClass="common\\models\\Post" \
  --enablePjax=1
```

Which generates a [[yii\widgets\Pjax|Pjax]] widget wrapping the [[yii\grid\GridView|GridView]] or [[yii\widgets\ListView|ListView]] widgets.

Further reading

- Rendering Data in Yii 2 with GridView and ListView by Arno Slatius.

Working with Client Scripts

Modern web applications, additionally to static HTML pages that are rendered and sent to the browser, contain JavaScript that is used to modify the page in the browser by manipulating existing elements or loading new content via AJAX. This section describes methods provided by Yii for adding JavaScript and CSS to a website as well as dynamically adjusting these.

Registering scripts

When working with the [[yii\web\View]] object you can dynamically register frontend scripts. There are two dedicated methods for this:

- [[yii\web\View::registerJs()|registerJs()]] for inline scripts
- [[yii\web\View::registerJsFile()|registerJsFile()]] for external scripts

Registering inline scripts

Inline scripts are useful for configuration, dynamically generated code and small snippets created by reusable frontend code contained in widgets. The [[yii\web\View::registerJs()|registerJs()]] method for adding these can be used as follows:

```
$this->registerJs(
    "$('#myButton').on('click', function() { alert('Button clicked!'); });",
    View::POS_READY,
    'my-button-handler'
);
```

The first argument is the actual JS code we want to insert into the page. It will be wrapped into a `<script>` tag. The second argument determines at which position the script should be inserted into the page. Possible values are:

- [[yii\web\View::POS_HEAD|View::POS_HEAD]] for head section.
- [[yii\web\View::POS_BEGIN|View::POS_BEGIN]] for right after opening `<body>`.
- [[yii\web\View::POS_END|View::POS_END]] for right before closing `</body>`.
- [[yii\web\View::POS_READY|View::POS_READY]] for executing code on the document `ready` event. This will automatically register [[yii\web\JqueryAsset|jQuery]] and wrap the code into the appropriate jQuery code. This is the default position.
- [[yii\web\View::POS_LOAD|View::POS_LOAD]] for executing code on the document

`load` event. Same as the above, this will also register [[yii\web\JqueryAsset|jQuery]] automatically.

The last argument is a unique script ID that is used to identify the script code block and replace an existing one with the same ID instead of adding a new one. If you don't provide it, the JS code itself will be used as the ID. It is used to avoid registration of the same code muliple times.

Registering script files

The arguments for [[yii\web\View::registerJsFile()|registerJsFile()]] are similar to those for [[yii\web\View::registerCssFile()|registerCssFile()]]. In the following example, we register the `main.js` file with the dependency on the [[yii\web\JqueryAsset]]. It means that the `main.js` file will be added AFTER `jquery.js`. Without such dependency specification, the relative order between `main.js` and `jquery.js` would be undefined and the code would not work.

An external script can be added like the following:

```
$this->registerJsFile(
    '@web/js/main.js',
    ['depends' => [\yii\web\JqueryAsset::className()]]
);
```

This will add a tag for the `/js/main.js` script located under the application base URL.

It is highly recommended to use asset bundles to register external JS files rather than [[yii\web\View::registerJsFile()|registerJsFile()]] because these allow better flexibility and more granular dependency configuration. Also using asset bundles allows you to combine and compress multiple JS files, which is desirable for high traffic websites.

Registering CSS

Similar to JavaScript, you can register CSS using [[yii\web\View::registerCss()|registerCss()]] or [[yii\web\View::registerCssFile()|registerCssFile()]]. The former registers a block of CSS code while the latter registers an external CSS file.

Registering inline CSS

```
$this->registerCss("body { background: #f00; }");
```

The code above will result in adding the following to the `<head>` section of the page:

```
<style>
body { background: #f00; }
</style>
```

If you want to specify additional properties of the style tag, pass an array of name-values to the second argument. The last argument is a unique ID that is used to identify the style block and make sure it is only added once in case the same style is registered from different places in the code.

Registering CSS files

A CSS file can be registered using the following:

```
$this->registerCssFile("@web/css/themes/black-and-white.css", [
    'depends' => [\yii\bootstrap\BootstrapAsset::className()],
    'media' => 'print',
], 'css-print-theme');
```

The above code will add a link to the `/css/themes/black-and-white.css` CSS file to the `<head>` section of the page.

- The first argument specifies the CSS file to be registered. The `@web` in this example is an alias for the applications base URL.
- The second argument specifies the HTML attributes for the resulting `<link>` tag. The option `depends` is specially handled. It specifies which asset bundles this CSS file depends on. In this case, the dependent asset bundle is [[yii\bootstrap\BootstrapAsset|BootstrapAsset]]. This means the CSS file will be added *after* the CSS files from [[yii\bootstrap\BootstrapAsset|BootstrapAsset]].
- The last argument specifies an ID identifying this CSS file. If it is not provided, the URL of the CSS file will be used instead.

It is highly recommended to use asset bundles to register external CSS files rather than [[yii\web\View::registerCssFile()|registerCssFile()]]. Using asset bundles allows you to combine and compress multiple CSS files, which is desirable for high traffic websites. It also provides more flexibility as all asset dependencies of your application are configured in one place.

Registering asset bundles

As was mentioned earlier it's recommended to use asset bundles instead of registering CSS and JavaScript files directly. You can get details on how to define asset bundles in the "Assets" section. As for using already defined asset bundles, it's very straightforward:

```
\frontend\assets\AppAsset::register($this);
```

In the above code, in the context of a view file, the `AppAsset` bundle is registered on the current view (represented by `$this`). When registering asset bundles from within a widget, you would pass the [[yii\base\Widget::$view|$view]] of the widget instead (`$this->view`).

Generating Dynamic Javascript

In view files often the HTML code is not written out directly but generated by some PHP code dependent on the variables of the view. In order for the generated HTML to be manipulated with Javascript, the JS code has to contain dynamic parts too, for example the IDs of the jQuery selectors.

To insert PHP variables into JS code, their values have to be escaped properly. Especially when the JS code is inserted into HTML instead of residing in a dedicated JS file. Yii provides the [[yii\helpers\Json::htmlEncode()|htmlEncode()]] method of the [[yii\helpers\Json|Json]] helper for this purpose. Its usage will be shown in the following examples.

Registering a global JavaScript configuration

In this example we use an array to pass global configuration parameters from the PHP part of the application to the JS frontend code.

```
$options = [
    'appName' => Yii::$app->name,
    'baseUrl' => Yii::$app->request->baseUrl,
    'language' => Yii::$app->language,
    // ...
];
$this->registerJs(
    "var yiiOptions = ".\yii\helpers\Json::htmlEncode($options).";",
    View::POS_HEAD,
    'yiiOptions'
);
```

The above code will register a `<script>`-tag containing the JavaScript variable definition, e.g.:

```
var yiiOptions = {"appName":"My Yii Application","baseUrl":"/basic/web","language":"en"
};
```

In your JavaScript code you can now access these like `yiiOptions.baseUrl` or `yiiOptions.language`.

Passing translated messages

You may encounter a case where your JavaScript needs to print a message reacting to some event. In an application that works with multiple languages this string has to be translated to the current application language. One way to achieve this is to use the message translation feature provided by Yii and passing the result to the JavaScript code.

```
$message = \yii\helpers\Json::htmlEncode(
    \Yii::t('app', 'Button clicked!')
);
$this->registerJs(<<<JS
    $('#myButton').on('click', function() { alert( $message ); });",
JS
);
```

The above example code uses PHP Heredoc syntax for better readability. This also enables better syntax highlighting in most IDEs so it is the preferred way of writing inline JavaScript, especially useful for code that is longer than a single line. The variable `$message` is created in PHP and thanks to [[yii\helpers\Json::htmlEncode|Json::htmlEncode]] it contains the string in valid JS syntax, which can be inserted into the JavaScript code to place the dynamic string in the function call to `alert()`.

> Note: When using Heredoc, be careful with variable naming in JS code as variables beginning with `$` may be interpreted as PHP variables which will be replaced by their content. The jQuery function in form of `$(` or `$.` is not interpreted as a PHP variable and can safely be used.

The `yii.js` script

> Note: This section has not been written yet. It should contain explanation of the functionality provided by `yii.js`:
>
> - Yii JavaScript Modules
> - CSRF param handling
> - `data-confirm` handler
> - `data-method` handler
> - script filtering
> - redirect handling

Theming

Theming is a way to replace a set of views with another without the need of touching the original view rendering code. You can use theming to systematically change the look and feel of an application.

To use theming, you should configure the [[yii\base\View::theme|theme]] property of the `view` application component. The property configures a [[yii\base\Theme]] object which governs how view files are being replaced. You should mainly specify the following properties of [[yii\base\Theme]]:

- [[yii\base\Theme::basePath]]: specifies the base directory that contains the themed resources (CSS, JS, images, etc.)
- [[yii\base\Theme::baseUrl]]: specifies the base URL of the themed resources.
- [[yii\base\Theme::pathMap]]: specifies the replacement rules of view files. More details will be given in the following subsections.

For example, if you call `$this->render('about')` in `SiteController`, you will be rendering the view file `@app/views/site/about.php`. However, if you enable theming in the following application configuration, the view file `@app/themes/basic/site/about.php` will be rendered, instead.

```
return [
    'components' => [
        'view' => [
            'theme' => [
                'basePath' => '@app/themes/basic',
                'baseUrl' => '@web/themes/basic',
                'pathMap' => [
                    '@app/views' => '@app/themes/basic',
                ],
            ],
        ],
    ],
];
```

> Info: Path aliases are supported by themes. When doing view replacement, path aliases will be turned into the actual file paths or URLs.

You can access the [[yii\base\Theme]] object through the [[yii\base\View::theme]] property. For example, in a view file, you can write the following code because `$this` refers to the view object:

```
$theme = $this->theme;

// returns: $theme->baseUrl . '/img/logo.gif'
$url = $theme->getUrl('img/logo.gif');

// returns: $theme->basePath . '/img/logo.gif'
$file = $theme->getPath('img/logo.gif');
```

The [[yii\base\Theme::pathMap]] property governs how view files should be replaced. It takes an array of key-value pairs, where the keys are the original view paths to be replaced and the values are the corresponding themed view paths. The replacement is based on partial match: if a view path starts with any key in the [[yii\base\Theme::pathMap|pathMap]] array, that matching part will be replaced with the corresponding array value. Using the above configuration example, because `@app/views/site/about.php` partially matches the key `@app/views`, it will be replaced as `@app/themes/basic/site/about.php`.

Theming Modules

In order to theme modules, [[yii\base\Theme::pathMap]] can be configured like the following:

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/modules' => '@app/themes/basic/modules', // <-- !!!
],
```

It will allow you to theme `@app/modules/blog/views/comment/index.php` into `@app/themes/basic/modules/blog/views/comment/index.php`.

Theming Widgets

In order to theme widgets, you can configure [[yii\base\Theme::pathMap]] in the following way:

```
'pathMap' => [
    '@app/views' => '@app/themes/basic',
    '@app/widgets' => '@app/themes/basic/widgets', // <-- !!!
],
```

This will allow you to theme `@app/widgets/currency/views/index.php` into `@app/themes/basic/widgets/currency/index.php`.

Theme Inheritance

Sometimes you may want to define a basic theme which contains a basic look and feel of the application, and then based on the current holiday, you may want to vary the look and feel slightly. You can achieve this goal using theme inheritance which is done by mapping a single view path to multiple targets. For example,

```
'pathMap' => [
    '@app/views' => [
        '@app/themes/christmas',
        '@app/themes/basic',
    ],
]
```

In this case, the view `@app/views/site/index.php` would be themed as either `@app/themes/christmas/site/index.php` or `@app/themes/basic/site/index.php`, depending on which themed file exists. If both themed files exist, the first one will take precedence. In practice, you would keep most themed view files in `@app/themes/basic` and customize some of them in `@app/themes/christmas`.

Authentication

Authentication is the process of verifying the identity of a user. It usually uses an identifier (e.g. a username or an email address) and a secret token (e.g. a password or an access token) to judge if the user is the one whom he claims as. Authentication is the basis of the login feature.

Yii provides an authentication framework which wires up various components to support login. To use this framework, you mainly need to do the following work:

- Configure the [[yii\web\User|user]] application component;
- Create a class that implements the [[yii\web\IdentityInterface]] interface.

Configuring [[yii\web\User]]

The [[yii\web\User|user]] application component manages the user authentication status. It requires you to specify an [[yii\web\User::identityClass|identity class]] which contains the actual authentication logic. In the following application configuration, the [[yii\web\User::identityClass|identity class]] for [[yii\web\User|user]] is configured to be `app\models\User` whose implementation is explained in the next subsection:

```
return [
    'components' => [
        'user' => [
            'identityClass' => 'app\models\User',
        ],
    ],
];
```

Implementing [[yii\web\IdentityInterface]]

The [[yii\web\User::identityClass|identity class]] must implement the [[yii\web\IdentityInterface]] which contains the following methods:

- [[yii\web\IdentityInterface::findIdentity()|findIdentity()]]: it looks for an instance of the identity class using the specified user ID. This method is used when you need to maintain the login status via session.
- [[yii\web\IdentityInterface::findIdentityByAccessToken()|findIdentityByAccessToken()]]: it looks for an instance of the identity class using the specified access token. This method

is used when you need to authenticate a user by a single secret token (e.g. in a stateless RESTful application).
- [[yii\web\IdentityInterface::getId()|getId()]]: it returns the ID of the user represented by this identity instance.
- [[yii\web\IdentityInterface::getAuthKey()|getAuthKey()]]: it returns a key used to verify cookie-based login. The key is stored in the login cookie and will be later compared with the server-side version to make sure the login cookie is valid.
- [[yii\web\IdentityInterface::validateAuthKey()|validateAuthKey()]]: it implements the logic for verifying the cookie-based login key.

If a particular method is not needed, you may implement it with an empty body. For example, if your application is a pure stateless RESTful application, you would only need to implement [[yii\web\IdentityInterface::findIdentityByAccessToken()|findIdentityByAccessToken()]] and [[yii\web\IdentityInterface::getId()|getId()]] while leaving all other methods with an empty body.

In the following example, an [[yii\web\User::identityClass|identity class]] is implemented as an Active Record class associated with the `user` database table.

```
<?php

use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function tableName()
    {
        return 'user';
    }

    /**
     * Finds an identity by the given ID.
     *
     * @param string|int $id the ID to be looked for
     * @return IdentityInterface|null the identity object that matches the given ID.
     */
    public static function findIdentity($id)
    {
        return static::findOne($id);
    }

    /**
     * Finds an identity by the given token.
     *
     * @param string $token the token to be looked for
     * @return IdentityInterface|null the identity object that matches the given token
.
```

```
     */
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }

    /**
     * @return int|string current user ID
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * @return string current user auth key
     */
    public function getAuthKey()
    {
        return $this->auth_key;
    }

    /**
     * @param string $authKey
     * @return bool if auth key is valid for current user
     */
    public function validateAuthKey($authKey)
    {
        return $this->getAuthKey() === $authKey;
    }
}
```

As explained previously, you only need to implement `getAuthKey()` and `validateAuthKey()` if your application uses cookie-based login feature. In this case, you may use the following code to generate an auth key for each user and store it in the `user` table:

```
class User extends ActiveRecord implements IdentityInterface
{
    ......

    public function beforeSave($insert)
    {
        if (parent::beforeSave($insert)) {
            if ($this->isNewRecord) {
                $this->auth_key = \Yii::$app->security->generateRandomString();
            }
            return true;
        }
        return false;
    }
}
```

> Note: Do not confuse the `User` identity class with [[yii\web\User]]. The former is the class implementing the authentication logic. It is often implemented as an Active Record class associated with some persistent storage for storing the user credential information. The latter is an application component class responsible for managing the user authentication state.

Using [[yii\web\User]]

You mainly use [[yii\web\User]] in terms of the `user` application component.

You can detect the identity of the current user using the expression `Yii::$app->user->identity`. It returns an instance of the [[yii\web\User::identityClass|identity class]] representing the currently logged-in user, or `null` if the current user is not authenticated (meaning a guest). The following code shows how to retrieve other authentication-related information from [[yii\web\User]]:

```
// the current user identity. `null` if the user is not authenticated.
$identity = Yii::$app->user->identity;

// the ID of the current user. `null` if the user not authenticated.
$id = Yii::$app->user->id;

// whether the current user is a guest (not authenticated)
$isGuest = Yii::$app->user->isGuest;
```

To login a user, you may use the following code:

```
// find a user identity with the specified username.
// note that you may want to check the password if needed
$identity = User::findOne(['username' => $username]);

// logs in the user
Yii::$app->user->login($identity);
```

The [[yii\web\User::login()]] method sets the identity of the current user to the [[yii\web\User]]. If session is [[yii\web\User::enableSession|enabled]], it will keep the identity in the session so that the user authentication status is maintained throughout the whole session. If cookie-based login (i.e. "remember me" login) is [[yii\web\User::enableAutoLogin|enabled]], it will also save the identity in a cookie so that the user authentication status can be recovered from the cookie as long as the cookie remains valid.

In order to enable cookie-based login, you need to configure [[yii\web\User::enableAutoLogin]] to be `true` in the application configuration. You also need to provide a duration time parameter when calling the [[yii\web\User::login()]] method.

To logout a user, simply call

```
Yii::$app->user->logout();
```

Note that logging out a user is only meaningful when session is enabled. The method will clean up the user authentication status from both memory and session. And by default, it will also destroy *all* user session data. If you want to keep the session data, you should call `Yii::$app->user->logout(false)`, instead.

Authentication Events

The [[yii\web\User]] class raises a few events during the login and logout processes.

- [[yii\web\User::EVENT_BEFORE_LOGIN|EVENT_BEFORE_LOGIN]]: raised at the beginning of [[yii\web\User::login()]]. If the event handler sets the [[yii\web\UserEvent::isValid|isValid]] property of the event object to be `false`, the login process will be cancelled.
- [[yii\web\User::EVENT_AFTER_LOGIN|EVENT_AFTER_LOGIN]]: raised after a successful login.
- [[yii\web\User::EVENT_BEFORE_LOGOUT|EVENT_BEFORE_LOGOUT]]: raised at the beginning of [[yii\web\User::logout()]]. If the event handler sets the [[yii\web\UserEvent::isValid|isValid]] property of the event object to be `false`, the logout

process will be cancelled.
- [[yii\web\User::EVENT_AFTER_LOGOUT|EVENT_AFTER_LOGOUT]]: raised after a successful logout.

You may respond to these events to implement features such as login audit, online user statistics. For example, in the handler for [[yii\web\User::EVENT_AFTER_LOGIN|EVENT_AFTER_LOGIN]], you may record the login time and IP address in the `user` table.

Authorization

Authorization is the process of verifying that a user has enough permission to do something. Yii provides two authorization methods: Access Control Filter (ACF) and Role-Based Access Control (RBAC).

Access Control Filter

Access Control Filter (ACF) is a simple authorization method implemented as [[yii\filters\AccessControl]] which is best used by applications that only need some simple access control. As its name indicates, ACF is an action filter that can be used in a controller or a module. While a user is requesting to execute an action, ACF will check a list of [[yii\filters\AccessControl::rules|access rules]] to determine if the user is allowed to access the requested action.

The code below shows how to use ACF in the `site` controller:

```
use yii\web\Controller;
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['login', 'logout', 'signup'],
                'rules' => [
                    [
                        'allow' => true,
                        'actions' => ['login', 'signup'],
                        'roles' => ['?'],
                    ],
                    [
                        'allow' => true,
                        'actions' => ['logout'],
                        'roles' => ['@'],
                    ],
                ],
            ],
        ];
    }
    // ...
}
```

In the code above ACF is attached to the `site` controller as a behavior. This is the typical way of using an action filter. The `only` option specifies that the ACF should only be applied to the `login`, `logout` and `signup` actions. All other actions in the `site` controller are not subject to the access control. The `rules` option lists the [[yii\filters\AccessRule|access rules]], which reads as follows:

- Allow all guest (not yet authenticated) users to access the `login` and `signup` actions. The `roles` option contains a question mark `?` which is a special token representing "guest users".
- Allow authenticated users to access the `logout` action. The `@` character is another special token representing "authenticated users".

ACF performs the authorization check by examining the access rules one by one from top to bottom until it finds a rule that matches the current execution context. The `allow` value of the matching rule will then be used to judge if the user is authorized or not. If none of the rules matches, it means the user is NOT authorized, and ACF will stop further action execution.

When ACF determines a user is not authorized to access the current action, it takes the following measure by default:

- If the user is a guest, it will call [[yii\web\User::loginRequired()]] to redirect the user browser to the login page.
- If the user is already authenticated, it will throw a [[yii\web\ForbiddenHttpException]].

You may customize this behavior by configuring the [[yii\filters\AccessControl::denyCallback]] property like the following:

```
[
    'class' => AccessControl::className(),
    ...
    'denyCallback' => function ($rule, $action) {
        throw new \Exception('You are not allowed to access this page');
    }
]
```

[[yii\filters\AccessRule|Access rules]] support many options. Below is a summary of the supported options. You may also extend [[yii\filters\AccessRule]] to create your own customized access rule classes.

- [[yii\filters\AccessRule::allow|allow]]: specifies whether this is an "allow" or "deny" rule.
- [[yii\filters\AccessRule::actions|actions]]: specifies which actions this rule matches. This should be an array of action IDs. The comparison is case-sensitive. If this option is empty or not set, it means the rule applies to all actions.
- [[yii\filters\AccessRule::controllers|controllers]]: specifies which controllers this rule matches. This should be an array of controller IDs. Each controller ID is prefixed with the module ID (if any). The comparison is case-sensitive. If this option is empty or not set, it means the rule applies to all controllers.
- [[yii\filters\AccessRule::roles|roles]]: specifies which user roles that this rule matches. Two special roles are recognized, and they are checked via [[yii\web\User::isGuest]]:
  - `?`: matches a guest user (not authenticated yet)
  - `@`: matches an authenticated user

  Using other role names will trigger the invocation of [[yii\web\User::can()]], which requires enabling RBAC (to be described in the next subsection). If this option is empty or not set, it means this rule applies to all roles.
- [[yii\filters\AccessRule::ips|ips]]: specifies which [[yii\web\Request::userIP|client IP addresses]] this rule matches. An IP address can contain the wildcard `*` at the end so that it matches IP addresses with the same prefix. For example, '192.168.*' matches all

IP addresses in the segment '192.168.'. If this option is empty or not set, it means this rule applies to all IP addresses.

- [[yii\filters\AccessRule::verbs|verbs]]: specifies which request method (e.g. `GET`, `POST`) this rule matches. The comparison is case-insensitive.
- [[yii\filters\AccessRule::matchCallback|matchCallback]]: specifies a PHP callable that should be called to determine if this rule should be applied.
- [[yii\filters\AccessRule::denyCallback|denyCallback]]: specifies a PHP callable that should be called when this rule will deny the access.

Below is an example showing how to make use of the `matchCallback` option, which allows you to write arbitrary access check logic:

```
use yii\filters\AccessControl;

class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['special-callback'],
                'rules' => [
                    [
                        'actions' => ['special-callback'],
                        'allow' => true,
                        'matchCallback' => function ($rule, $action) {
                            return date('d-m') === '31-10';
                        }
                    ],
                ],
            ],
        ];
    }

    // Match callback called! This page can be accessed only each October 31st
    public function actionSpecialCallback()
    {
        return $this->render('happy-halloween');
    }
}
```

Role Based Access Control (RBAC)

Role-Based Access Control (RBAC) provides a simple yet powerful centralized access control. Please refer to the Wikipedia for details about comparing RBAC with other more traditional access control schemes.

Yii implements a General Hierarchical RBAC, following the NIST RBAC model. It provides the RBAC functionality through the [[yii\rbac\ManagerInterface|authManager]] application component.

Using RBAC involves two parts of work. The first part is to build up the RBAC authorization data, and the second part is to use the authorization data to perform access check in places where it is needed.

To facilitate our description next, we will first introduce some basic RBAC concepts.

Basic Concepts

A role represents a collection of *permissions* (e.g. creating posts, updating posts). A role may be assigned to one or multiple users. To check if a user has a specified permission, we may check if the user is assigned with a role that contains that permission.

Associated with each role or permission, there may be a *rule*. A rule represents a piece of code that will be executed during access check to determine if the corresponding role or permission applies to the current user. For example, the "update post" permission may have a rule that checks if the current user is the post creator. During access checking, if the user is NOT the post creator, he/she will be considered not having the "update post" permission.

Both roles and permissions can be organized in a hierarchy. In particular, a role may consist of other roles or permissions; and a permission may consist of other permissions. Yii implements a *partial order* hierarchy which includes the more special *tree* hierarchy. While a role can contain a permission, it is not `true` vice versa.

Configuring RBAC

Before we set off to define authorization data and perform access checking, we need to configure the [[yii\base\Application::authManager|authManager]] application component. Yii provides two types of authorization managers: [[yii\rbac\PhpManager]] and [[yii\rbac\DbManager]]. The former uses a PHP script file to store authorization data, while the latter stores authorization data in a database. You may consider using the former if your application does not require very dynamic role and permission management.

Using `PhpManager`

The following code shows how to configure the `authManager` in the application configuration using the [[yii\rbac\PhpManager]] class:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

The `authManager` can now be accessed via `\Yii::$app->authManager`.

By default, [[yii\rbac\PhpManager]] stores RBAC data in files under `@app/rbac` directory. Make sure the directory and all the files in it are writable by the Web server process if permissions hierarchy needs to be changed online.

Using `DbManager`

The following code shows how to configure the `authManager` in the application configuration using the [[yii\rbac\DbManager]] class:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\DbManager',
        ],
        // ...
    ],
];
```

> Note: If you are using yii2-basic-app template, there is a `config/console.php` configuration file where the `authManager` needs to be declared additionally to `config/web.php`. In case of yii2-advanced-app the `authManager` should be declared only once in `common/config/main.php`.

`DbManager` uses four database tables to store its data:

- [[yii\rbac\DbManager::$itemTable|itemTable]]: the table for storing authorization items. Defaults to "auth_item".
- [[yii\rbac\DbManager::$itemChildTable|itemChildTable]]: the table for storing authorization item hierarchy. Defaults to "auth_item_child".

- [[yii\rbac\DbManager::$assignmentTable|assignmentTable]]: the table for storing authorization item assignments. Defaults to "auth_assignment".
- [[yii\rbac\DbManager::$ruleTable|ruleTable]]: the table for storing rules. Defaults to "auth_rule".

Before you can go on you need to create those tables in the database. To do this, you can use the migration stored in `@yii/rbac/migrations`:

```
yii migrate --migrationPath=@yii/rbac/migrations
```

Read more about working with migrations from different namespaces in Separated Migrations section.

The `authManager` can now be accessed via `\Yii::$app->authManager`.

Building Authorization Data

Building authorization data is all about the following tasks:

- defining roles and permissions;
- establishing relations among roles and permissions;
- defining rules;
- associating rules with roles and permissions;
- assigning roles to users.

Depending on authorization flexibility requirements the tasks above could be done in different ways.

If your permissions hierarchy doesn't change at all and you have a fixed number of users you can create a console command that will initialize authorization data once via APIs offered by `authManager`:

```php
<?php
namespace app\commands;

use Yii;
use yii\console\Controller;

class RbacController extends Controller
{
    public function actionInit()
    {
        $auth = Yii::$app->authManager;

        // add "createPost" permission
        $createPost = $auth->createPermission('createPost');
        $createPost->description = 'Create a post';
        $auth->add($createPost);

        // add "updatePost" permission
        $updatePost = $auth->createPermission('updatePost');
        $updatePost->description = 'Update post';
        $auth->add($updatePost);

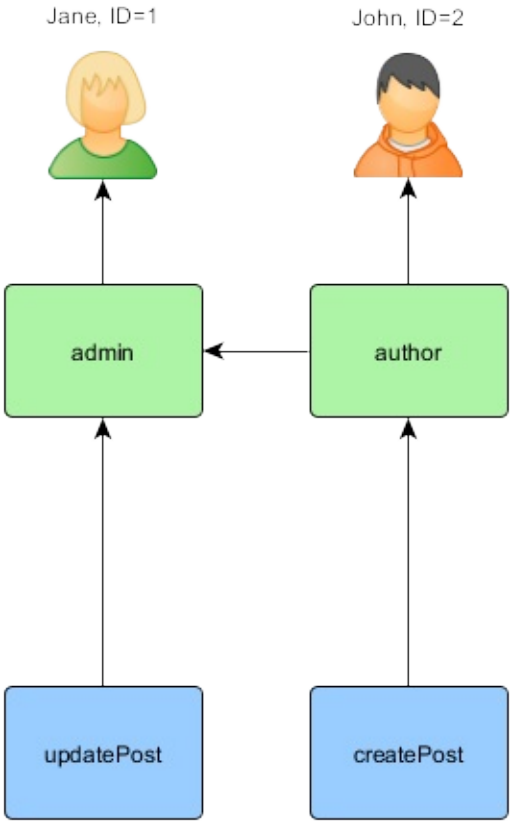
        // add "author" role and give this role the "createPost" permission
        $author = $auth->createRole('author');
        $auth->add($author);
        $auth->addChild($author, $createPost);

        // add "admin" role and give this role the "updatePost" permission
        // as well as the permissions of the "author" role
        $admin = $auth->createRole('admin');
        $auth->add($admin);
        $auth->addChild($admin, $updatePost);
        $auth->addChild($admin, $author);

        // Assign roles to users. 1 and 2 are IDs returned by IdentityInterface::getId
()
        // usually implemented in your User model.
        $auth->assign($author, 2);
        $auth->assign($admin, 1);
    }
}
```

> Note: If you are using advanced template, you need to put your `RbacController` inside `console/controllers` directory and change namespace to `console\controllers`.

After executing the command with `yii rbac/init` we'll get the following hierarchy:



Author can create post, admin can update post and do everything author can.

If your application allows user signup you need to assign roles to these new users once. For example, in order for all signed up users to become authors in your advanced project template you need to modify `frontend\models\SignupForm::signup()` as follows:

```
public function signup()
{
    if ($this->validate()) {
        $user = new User();
        $user->username = $this->username;
        $user->email = $this->email;
        $user->setPassword($this->password);
        $user->generateAuthKey();
        $user->save(false);

        // the following three lines were added:
        $auth = \Yii::$app->authManager;
        $authorRole = $auth->getRole('author');
        $auth->assign($authorRole, $user->getId());

        return $user;
    }

    return null;
}
```

For applications that require complex access control with dynamically updated authorization data, special user interfaces (i.e. admin panel) may need to be developed using APIs offered by `authManager`.

Using Rules

As aforementioned, rules add additional constraint to roles and permissions. A rule is a class extending from [[yii\rbac\Rule]]. It must implement the [[yii\rbac\Rule::execute()|execute()]] method. In the hierarchy we've created previously author cannot edit his own post. Let's fix it. First we need a rule to verify that the user is the post author:

```
namespace app\rbac;

use yii\rbac\Rule;

/**
 * Checks if authorID matches user passed via params
 */
class AuthorRule extends Rule
{
    public $name = 'isAuthor';

    /**
     * @param string|int $user the user ID.
     * @param Item $item the role or permission that this rule is associated with
     * @param array $params parameters passed to ManagerInterface::checkAccess().
     * @return bool a value indicating whether the rule permits the role or permission
 it is associated with.
     */
    public function execute($user, $item, $params)
    {
        return isset($params['post']) ? $params['post']->createdBy == $user : false;
    }
}
```

The rule above checks if the `post` is created by `$user`. We'll create a special permission `updateOwnPost` in the command we've used previously:

```
$auth = Yii::$app->authManager;

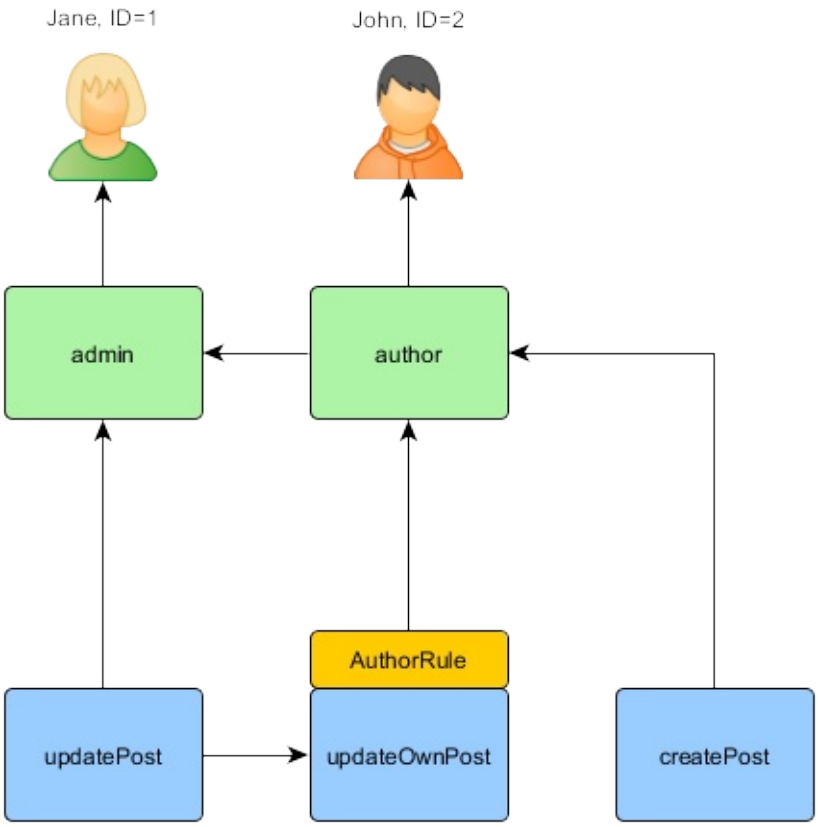
// add the rule
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

// add the "updateOwnPost" permission and associate the rule with it.
$updateOwnPost = $auth->createPermission('updateOwnPost');
$updateOwnPost->description = 'Update own post';
$updateOwnPost->ruleName = $rule->name;
$auth->add($updateOwnPost);

// "updateOwnPost" will be used from "updatePost"
$auth->addChild($updateOwnPost, $updatePost);

// allow "author" to update their own posts
$auth->addChild($author, $updateOwnPost);
```

Now we have got the following hierarchy:



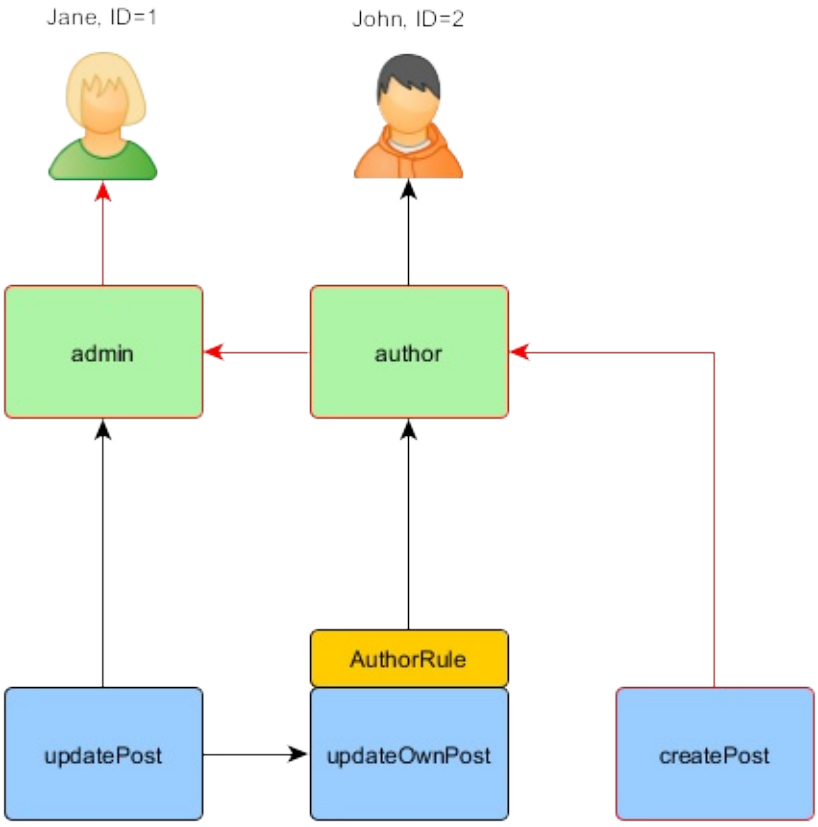
Access Check

With the authorization data ready, access check is as simple as a call to the [[yii\rbac\ManagerInterface::checkAccess()]] method. Because most access check is about the current user, for convenience Yii provides a shortcut method [[yii\web\User::can()]],

which can be used like the following:

```
if (\Yii::$app->user->can('createPost')) {
    // create post
}
```

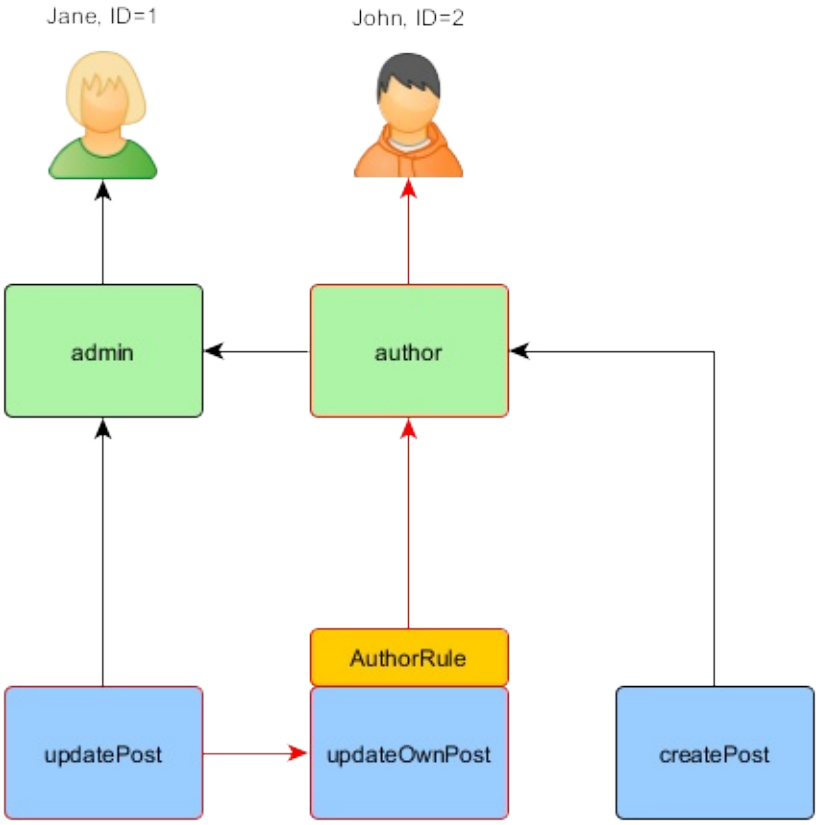
If the current user is Jane with `ID=1` we are starting at `createPost` and trying to get to `Jane`:



In order to check if a user can update a post, we need to pass an extra parameter that is required by `AuthorRule` described before:

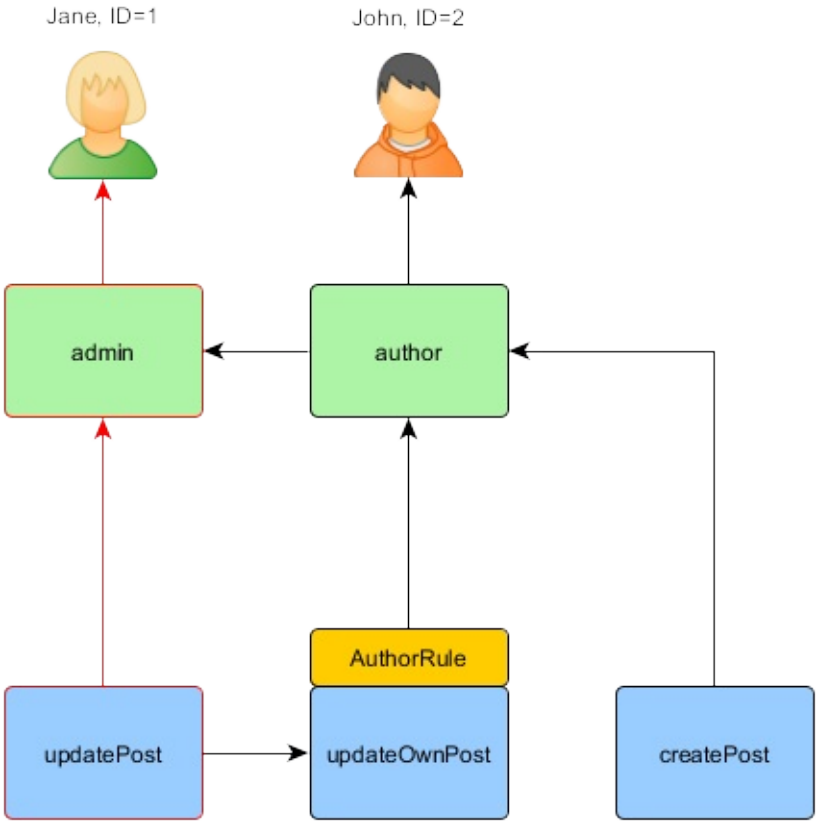
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {
    // update post
}
```

Here is what happens if the current user is John:



We are starting with the `updatePost` and going through `updateOwnPost`. In order to pass the access check, `AuthorRule` should return `true` from its `execute()` method. The method receives its `$params` from the `can()` method call so the value is `['post' => $post]`. If everything is fine, we will get to `author` which is assigned to John.

In case of Jane it is a bit simpler since she is an admin:



Inside your controller there are a few ways to implement authorization. If you want granular permissions that separate access to adding and deleting, then you need to check access for each action. You can either use the above condition in each action method, or use [[yii\filters\AccessControl]]:

```
public function behaviors()
{
    return [
        'access' => [
            'class' => AccessControl::className(),
            'rules' => [
                [
                    'allow' => true,
                    'actions' => ['index'],
                    'roles' => ['managePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['view'],
                    'roles' => ['viewPost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['create'],
                    'roles' => ['createPost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['update'],
                    'roles' => ['updatePost'],
                ],
                [
                    'allow' => true,
                    'actions' => ['delete'],
                    'roles' => ['deletePost'],
                ],
            ],
        ],
    ];
}
```

If all the CRUD operations are managed together then it's a good idea to use a single permission, like `managePost`, and check it in [[yii\web\Controller::beforeAction()]].

Using Default Roles

A default role is a role that is *implicitly* assigned to *all* users. The call to [[yii\rbac\ManagerInterface::assign()]] is not needed, and the authorization data does not contain its assignment information.

A default role is usually associated with a rule which determines if the role applies to the user being checked.

Default roles are often used in applications which already have some sort of role assignment. For example, an application may have a "group" column in its user table to represent which privilege group each user belongs to. If each privilege group can be mapped to an RBAC role, you can use the default role feature to automatically assign each user to an RBAC role. Let's use an example to show how this can be done.

Assume in the user table, you have a `group` column which uses 1 to represent the administrator group and 2 the author group. You plan to have two RBAC roles `admin` and `author` to represent the permissions for these two groups, respectively. You can set up the RBAC data as follows,

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * Checks if user group matches
 */
class UserGroupRule extends Rule
{
    public $name = 'userGroup';

    public function execute($user, $item, $params)
    {
        if (!Yii::$app->user->isGuest) {
            $group = Yii::$app->user->identity->group;
            if ($item->name === 'admin') {
                return $group == 1;
            } elseif ($item->name === 'author') {
                return $group == 1 || $group == 2;
            }
        }
        return false;
    }
}

$auth = Yii::$app->authManager;

$rule = new \app\rbac\UserGroupRule;
$auth->add($rule);

$author = $auth->createRole('author');
$author->ruleName = $rule->name;
$auth->add($author);
// ... add permissions as children of $author ...

$admin = $auth->createRole('admin');
$admin->ruleName = $rule->name;
$auth->add($admin);
$auth->addChild($admin, $author);
// ... add permissions as children of $admin ...
```

Note that in the above, because "author" is added as a child of "admin", when you implement the `execute()` method of the rule class, you need to respect this hierarchy as well. That is why when the role name is "author", the `execute()` method will return `true` if the user group is either 1 or 2 (meaning the user is in either "admin" group or "author" group).

Next, configure `authManager` by listing the two roles in [[yii\rbac\BaseManager::$defaultRoles]]:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
            'defaultRoles' => ['admin', 'author'],
        ],
        // ...
    ],
];
```

Now if you perform an access check, both of the `admin` and `author` roles will be checked by evaluating the rules associated with them. If the rule returns `true`, it means the role applies to the current user. Based on the above rule implementation, this means if the `group` value of a user is 1, the `admin` role would apply to the user; and if the `group` value is 2, the `author` role would apply.

Working with Passwords

Most developers know that passwords cannot be stored in plain text, but many developers believe it's still safe to hash passwords using `md5` or `sha1`. There was a time when using the aforementioned hashing algorithms was sufficient, but modern hardware makes it possible to reverse such hashes and even stronger ones very quickly using brute force attacks.

In order to provide increased security for user passwords, even in the worst case scenario (your application is breached), you need to use a hashing algorithm that is resilient against brute force attacks. The best current choice is `bcrypt`. In PHP, you can create a `bcrypt` hash using the crypt function. Yii provides two helper functions which make using `crypt` to securely generate and verify hashes easier.

When a user provides a password for the first time (e.g., upon registration), the password needs to be hashed:

```
$hash = Yii::$app->getSecurity()->generatePasswordHash($password);
```

The hash can then be associated with the corresponding model attribute, so it can be stored in the database for later use.

When a user attempts to log in, the submitted password must be verified against the previously hashed and stored password:

```
if (Yii::$app->getSecurity()->validatePassword($password, $hash)) {
    // all good, logging user in
} else {
    // wrong password
}
```

Security best practices

Below we'll review common security principles and describe how to avoid threats when developing applications using Yii.

Basic principles

There are two main principles when it comes to security no matter which application is being developed:

1. Filter input.
2. Escape output.

Filter input

Filter input means that input should never be considered safe and you should always check if the value you've got is actually among allowed ones. For example, if we know that sorting could be done by three fields `title`, `created_at` and `status` and the field could be supplied via user input, it's better to check the value we've got right where we're receiving it. In terms of basic PHP that would look like the following:

```
$sortBy = $_GET['sort'];
if (!in_array($sortBy, ['title', 'created_at', 'status'])) {
    throw new Exception('Invalid sort value.');
}
```

In Yii, most probably you'll use form validation to do alike checks.

Escape output

Escape output means that depending on context where we're using data it should be escaped i.e. in context of HTML you should escape `<`, `>` and alike special characters. In context of JavaScript or SQL it will be different set of characters. Since it's error-prone to escape everything manually Yii provides various tools to perform escaping for different contexts.

Avoiding SQL injections

SQL injection happens when query text is formed by concatenating unescaped strings such as the following:

```
$username = $_GET['username'];
$sql = "SELECT * FROM user WHERE username = '$username'";
```

Instead of supplying correct username attacker could give your applications something like `'; DROP TABLE user; --`. Resulting SQL will be the following:

```
SELECT * FROM user WHERE username = ''; DROP TABLE user; --'
```

This is valid query that will search for users with empty username and then will drop `user` table most probably resulting in broken website and data loss (you've set up regular backups, right?).

In Yii most of database querying happens via Active Record which properly uses PDO prepared statements internally. In case of prepared statements it's not possible to manipulate query as was demonstrated above.

Still, sometimes you need raw queries or query builder. In this case you should use safe ways of passing data. If data is used for column values it's preferred to use prepared statements:

```
// query builder
$userIDs = (new Query())
    ->select('id')
    ->from('user')
    ->where('status=:status', [':status' => $status])
    ->all();

// DAO
$userIDs = $connection
    ->createCommand('SELECT id FROM user where status=:status')
    ->bindValues([':status' => $status])
    ->queryColumn();
```

If data is used to specify column names or table names the best thing to do is to allow only predefined set of values:

```
function actionList($orderBy = null)
{
    if (!in_array($orderBy, ['name', 'status'])) {
        throw new BadRequestHttpException('Only name and status are allowed to order b
y.')
    }

    // ...
}
```

In case it's not possible, table and column names should be escaped. Yii has special syntax for such escaping which allows doing it the same way for all databases it supports:

```
$sql = "SELECT COUNT([[$column]]) FROM {{table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

You can get details about the syntax in Quoting Table and Column Names.

Avoiding XSS

XSS or cross-site scripting happens when output isn't escaped properly when outputting HTML to the browser. For example, if user can enter his name and instead of `Alexander` he enters `<script>alert('Hello!');</script>`, every page that outputs user name without escaping it will execute JavaScript `alert('Hello!');` resulting in alert box popping up in a browser. Depending on website instead of innocent alert such script could send messages using your name or even perform bank transactions.

Avoiding XSS is quite easy in Yii. There are generally two cases:

1. You want data to be outputted as plain text.
2. You want data to be outputted as HTML.

If all you need is plain text then escaping is as easy as the following:

```
<?= \yii\helpers\Html::encode($username) ?>
```

If it should be HTML we could get some help from HtmlPurifier:

```
<?= \yii\helpers\HtmlPurifier::process($description) ?>
```

Note that HtmlPurifier processing is quite heavy so consider adding caching.

Avoiding CSRF

CSRF is an abbreviation for cross-site request forgery. The idea is that many applications assume that requests coming from a user browser are made by the user himself. It could be `false`.

For example, `an.example.com` website has `/logout` URL that, when accessed using a simple GET, logs user out. As long as it's requested by the user itself everything is OK but one day bad guys are somehow posting `<img src="http://an.example.com/logout">` on a forum user visits frequently. Browser doesn't make any difference between requesting an image or requesting a page so when user opens a page with such `img` tag, the browser will send the GET request to that URL, and the user will be logged out from `an.example.com`.

That's the basic idea. One can say that logging user out is nothing serious, but bad guys can do much more, using this idea. Imagine that some website has an URL `http://an.example.com/purse/transfer?to=anotherUser&amount=2000`. Accessing it using GET request, causes transfer of $2000 from authorized user account to user `anotherUser`. We know, that browser will always send GET request to load an image, so we can modify code to accept only POST requests on that URL. Unfortunately, this will not save us, because an attacker can put some JavaScript code instead of `<img>` tag, which allows to send POST requests on that URL.

In order to avoid CSRF you should always:

1. Follow HTTP specification i.e. GET should not change application state.
2. Keep Yii CSRF protection enabled.

Sometimes you need to disable CSRF validation per controller and/or action. It could be achieved by setting its property:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public $enableCsrfValidation = false;

    public function actionIndex()
    {
        // CSRF validation will not be applied to this and other actions
    }

}
```

To disable CSRF validation per custom actions you can do:

```
namespace app\controllers;

use yii\web\Controller;

class SiteController extends Controller
{
    public function beforeAction($action)
    {
        // ...set `$this->enableCsrfValidation` here based on some conditions...
        // call parent method that will check CSRF if such property is `true`.
        return parent::beforeAction($action);
    }
}
```

Avoiding file exposure

By default server webroot is meant to be pointed to `web` directory where `index.php` is. In case of shared hosting environments it could be impossible to achieve so we'll end up with all the code, configs and logs in server webroot.

If it's the case don't forget to deny access to everything except `web`. If it can't be done consider hosting your application elsewhere.

Avoiding debug info and tools at production

In debug mode Yii shows quite verbose errors which are certainly helpful for development. The thing is that these verbose errors are handy for attacker as well since these could reveal database structure, configuration values and parts of your code. Never run production applications with `YII_DEBUG` set to `true` in your `index.php`.

You should never enable Gii at production. It could be used to get information about database structure, code and to simply rewrite code with what's generated by Gii.

Debug toolbar should be avoided at production unless really necessary. It exposes all the application and config details possible. If you absolutely need it check twice that access is properly restricted to your IP only.

Using secure connection over TLS

Yii provides features that rely on cookies and/or PHP sessions. These can be vulnerable in case your connection is compromised. The risk is reduced if the app uses secure connection via TLS.

Please refer to your webserver documentation for instructions on how to configure it. You may also check example configs provided by H5BP project:

- Nginx
- Apache.
- IIS.
- Lighttpd.

Secure Server configuration

The purpose of this section is to highlight risks that need to be considered when creating a server configuration for serving a Yii based website. Besides the points covered here there may be other security related configuration options to be considered, so do not consider this section to be complete.

Avoiding `Host` -header attacks

Classes like [[yii\web\UrlManager]] and [[yii\helpers\Url]] may use the [[yii\web\Request::getHostInfo()|currently requested host name]] for generating links. If the webserver is configured to serve the same site independent of the value of the `Host` header, this information may not be reliable and may be faked by the user sending the HTTP request. In such situations you should either fix your webserver configuration to serve the site only for specified host names or explicitly set or filter the value by setting the [[yii\web\Request::setHostInfo()|hostInfo]] property of the `request` application component.

For more information about the server configuration, please refer to the documentation of your webserver:

- Apache 2: http://httpd.apache.org/docs/trunk/vhosts/examples.html#defaultallports
- Nginx: https://www.nginx.com/resources/wiki/start/topics/examples/server_blocks/

If you don't have access to the server configuration, you can setup [[yii\filters\HostControl]] filter at application level in order to protect against such kind of attack:

```
// Web Application configuration file
return [
    'as hostControl' => [
        'class' => 'yii\filters\HostControl',
        'allowedHosts' => [
            'example.com',
            '*.example.com',
        ],
        'fallbackHostInfo' => 'https://example.com',
    ],
    // ...
];
```

> Note: you should always prefer web server configuration for 'host header attack' protection instead of the filter usage. [[yii\filters\HostControl]] should be used only if server configuration setup is unavailable.

Caching

Caching is a cheap and effective way to improve the performance of a Web application. By storing relatively static data in cache and serving it from cache when requested, the application saves the time that would be required to generate the data from scratch every time.

Caching can occur at different levels and places in a Web application. On the server-side, at the lower level, cache may be used to store basic data, such as a list of most recent article information fetched from database; and at the higher level, cache may be used to store fragments or whole of Web pages, such as the rendering result of the most recent articles. On the client-side, HTTP caching may be used to keep most recently visited page content in the browser cache.

Yii supports all these caching mechanisms:

- Data caching
- Fragment caching
- Page caching
- HTTP caching

Data Caching

Data caching is about storing some PHP variables in cache and retrieving it later from cache. It is also the foundation for more advanced caching features, such as query caching and page caching.

The following code is a typical usage pattern of data caching, where `$cache` refers to a cache component:

```
// try retrieving $data from cache
$data = $cache->get($key);

if ($data === false) {
    // $data is not found in cache, calculate it from scratch
    $data = $this->calculateSomething();

    // store $data in cache so that it can be retrieved next time
    $cache->set($key, $data);
}

// $data is available here
```

Since version 2.0.11, cache component provides [[yii\caching\Cache::getOrSet()|getOrSet()]] method that simplifies code for data getting, calculating and storing. The following code does exactly the same as the previous example:

```
$data = $cache->getOrSet($key, function () {
    return $this->calculateSomething();
});
```

When cache has data associated with the `$key`, the cached value will be returned. Otherwise, the passed anonymous function will be executed to calculate the value that will be cached and returned.

If the anonymous function requires some data from the outer scope, you can pass it with the `use` statement. For example:

```
$user_id = 42;
$data = $cache->getOrSet($key, function () use ($user_id) {
    return $this->calculateSomething($user_id);
});
```

> Note: [[yii\caching\Cache::getOrSet()|getOrSet()]] method supports duration and dependencies as well. See Cache Expiration and Cache Dependencies to know more.

Cache Components

Data caching relies on the so-called *cache components* which represent various cache storage, such as memory, files, databases.

Cache components are usually registered as application components so that they can be globally configurable and accessible. The following code shows how to configure the `cache` application component to use memcached with two cache servers:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\MemCache',
        'servers' => [
            [
                'host' => 'server1',
                'port' => 11211,
                'weight' => 100,
            ],
            [
                'host' => 'server2',
                'port' => 11211,
                'weight' => 50,
            ],
        ],
    ],
],
```

You can then access the above cache component using the expression `Yii::$app->cache`.

Because all cache components support the same set of APIs, you can swap the underlying cache component with a different one by reconfiguring it in the application configuration without modifying the code that uses the cache. For example, you can modify the above configuration to use [[yii\caching\ApcCache|APC cache]]:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
    ],
],
```

> Tip: You can register multiple cache application components. The component named `cache` is used by default by many cache-dependent classes (e.g. [[yii\web\UrlManager]]).

Supported Cache Storage

Yii supports a wide range of cache storage. The following is a summary:

- [[yii\caching\ApcCache]]: uses PHP APC extension. This option can be considered as the fastest one when dealing with cache for a centralized thick application (e.g. one server, no dedicated load balancers, etc.).
- [[yii\caching\DbCache]]: uses a database table to store cached data. To use this cache, you must create a table as specified in [[yii\caching\DbCache::cacheTable]].
- [[yii\caching\DummyCache]]: serves as a cache placeholder which does no real caching. The purpose of this component is to simplify the code that needs to check the availability of cache. For example, during development or if the server doesn't have actual cache support, you may configure a cache component to use this cache. When an actual cache support is enabled, you can switch to use the corresponding cache component. In both cases, you may use the same code `Yii::$app->cache->get($key)` to attempt retrieving data from the cache without worrying that `Yii::$app->cache` might be `null`.
- [[yii\caching\FileCache]]: uses standard files to store cached data. This is particularly suitable to cache large chunk of data, such as page content.
- [[yii\caching\MemCache]]: uses PHP memcache and memcached extensions. This option can be considered as the fastest one when dealing with cache in a distributed applications (e.g. with several servers, load balancers, etc.)
- [[yii\redis\Cache]]: implements a cache component based on Redis key-value store (redis version 2.6.12 or higher is required).
- [[yii\caching\WinCache]]: uses PHP WinCache (see also) extension.
- [[yii\caching\XCache]]: uses PHP XCache extension.
- [[yii\caching\ZendDataCache]]: uses Zend Data Cache as the underlying caching medium.

> Tip: You may use different cache storage in the same application. A common strategy is to use memory-based cache storage to store data that is small but constantly used (e.g. statistical data), and use file-based or database-based cache storage to store data that is big and less frequently used (e.g. page content).

Cache APIs

All cache components have the same base class [[yii\caching\Cache]] and thus support the following APIs:

- [[yii\caching\Cache::get()|get()]]: retrieves a data item from cache with a specified key. A `false` value will be returned if the data item is not found in the cache or is expired/invalidated.
- [[yii\caching\Cache::set()|set()]]: stores a data item identified by a key in cache.
- [[yii\caching\Cache::add()|add()]]: stores a data item identified by a key in cache if the key is not found in the cache.
- [[yii\caching\Cache::getOrSet()|getOrSet()]]: retrieves a data item from cache with a specified key or executes passed callback, stores return of the callback in a cache by a key and returns that data.
- [[yii\caching\Cache::multiGet()|multiGet()]]: retrieves multiple data items from cache with the specified keys.
- [[yii\caching\Cache::multiSet()|multiSet()]]: stores multiple data items in cache. Each item is identified by a key.
- [[yii\caching\Cache::multiAdd()|multiAdd()]]: stores multiple data items in cache. Each item is identified by a key. If a key already exists in the cache, the data item will be skipped.
- [[yii\caching\Cache::exists()|exists()]]: returns a value indicating whether the specified key is found in the cache.
- [[yii\caching\Cache::delete()|delete()]]: removes a data item identified by a key from the cache.
- [[yii\caching\Cache::flush()|flush()]]: removes all data items from the cache.

> Note: Do not cache a `false` boolean value directly because the [[yii\caching\Cache::get()|get()]] method uses `false` return value to indicate the data item is not found in the cache. You may put `false` in an array and cache this array instead to avoid this problem.

Some cache storage, such as MemCache, APC, support retrieving multiple cached values in a batch mode, which may reduce the overhead involved in retrieving cached data. The APIs [[yii\caching\Cache::multiGet()|multiGet()]] and [[yii\caching\Cache::multiAdd()|multiAdd()]] are provided to exploit this feature. In case the underlying cache storage does not support this feature, it will be simulated.

Because [[yii\caching\Cache]] implements `ArrayAccess`, a cache component can be used like an array. The following are some examples:

```
$cache['var1'] = $value1;  // equivalent to: $cache->set('var1', $value1);
$value2 = $cache['var2'];  // equivalent to: $value2 = $cache->get('var2');
```

Cache Keys

Each data item stored in cache is uniquely identified by a key. When you store a data item in cache, you have to specify a key for it. Later when you retrieve the data item from cache, you should provide the corresponding key.

You may use a string or an arbitrary value as a cache key. When a key is not a string, it will be automatically serialized into a string.

A common strategy of defining a cache key is to include all determining factors in terms of an array. For example, [[yii\db\Schema]] uses the following key to cache schema information about a database table:

```
[
    __CLASS__,              // schema class name
    $this->db->dsn,         // DB connection data source name
    $this->db->username,    // DB connection login user
    $name,                  // table name
];
```

As you can see, the key includes all necessary information needed to uniquely specify a database table.

> Note: Values stored in cache via [[yii\caching\Cache::multiSet()|multiSet()]] or [[yii\caching\Cache::multiAdd()|multiAdd()]] can have only string or integer keys. If you need to set more complex key store the value separately via [[yii\caching\Cache::set()|set()]] or [[yii\caching\Cache::add()|add()]].

When the same cache storage is used by different applications, you should specify a unique cache key prefix for each application to avoid conflicts of cache keys. This can be done by configuring the [[yii\caching\Cache::keyPrefix]] property. For example, in the application configuration you can write the following code:

```
'components' => [
    'cache' => [
        'class' => 'yii\caching\ApcCache',
        'keyPrefix' => 'myapp',       // a unique cache key prefix
    ],
],
```

To ensure interoperability, only alphanumeric characters should be used.

Cache Expiration

A data item stored in a cache will remain there forever unless it is removed because of some caching policy enforcement (e.g. caching space is full and the oldest data are removed). To change this behavior, you can provide an expiration parameter when calling [[yii\caching\Cache::set()|set()]] to store a data item. The parameter indicates for how many seconds the data item can remain valid in the cache. When you call [[yii\caching\Cache::get()|get()]] to retrieve the data item, if it has passed the expiration time, the method will return `false`, indicating the data item is not found in the cache. For example,

```
// keep the data in cache for at most 45 seconds
$cache->set($key, $data, 45);

sleep(50);

$data = $cache->get($key);
if ($data === false) {
    // $data is expired or is not found in the cache
}
```

Since 2.0.11 you may set [[yii\caching\Cache::$defaultDuration|defaultDuration]] value in your cache component configuration if you prefer a custom cache duration over the default unlimited duration. This will allow you not to pass custom `duration` parameter to [[yii\caching\Cache::set()|set()]] each time.

Cache Dependencies

Besides expiration setting, cached data item may also be invalidated by changes of the so-called *cache dependencies*. For example, [[yii\caching\FileDependency]] represents the dependency of a file's modification time. When this dependency changes, it means the corresponding file is modified. As a result, any outdated file content found in the cache should be invalidated and the [[yii\caching\Cache::get()|get()]] call should return `false`.

Cache dependencies are represented as objects of [[yii\caching\Dependency]] descendant classes. When you call [[yii\caching\Cache::set()|set()]] to store a data item in the cache, you can pass along an associated cache dependency object. For example,

```
// Create a dependency on the modification time of file example.txt.
$dependency = new \yii\caching\FileDependency(['fileName' => 'example.txt']);

// The data will expire in 30 seconds.
// It may also be invalidated earlier if example.txt is modified.
$cache->set($key, $data, 30, $dependency);

// The cache will check if the data has expired.
// It will also check if the associated dependency was changed.
// It will return false if any of these conditions are met.
$data = $cache->get($key);
```

Below is a summary of the available cache dependencies:

- [[yii\caching\ChainedDependency]]: the dependency is changed if any of the dependencies on the chain is changed.
- [[yii\caching\DbDependency]]: the dependency is changed if the query result of the specified SQL statement is changed.
- [[yii\caching\ExpressionDependency]]: the dependency is changed if the result of the specified PHP expression is changed.
- [[yii\caching\FileDependency]]: the dependency is changed if the file's last modification time is changed.
- [[yii\caching\TagDependency]]: associates a cached data item with one or multiple tags. You may invalidate the cached data items with the specified tag(s) by calling [[yii\caching\TagDependency::invalidate()]].

> Note: Avoid using [[yii\caching\Cache::exists()|exists()]] method along with dependencies. It does not check whether the dependency associated with the cached data, if there is any, has changed. So a call to [[yii\caching\Cache::get()|get()]] may return `false` while [[yii\caching\Cache::exists()|exists()]] returns `true`.

Query Caching

Query caching is a special caching feature built on top of data caching. It is provided to cache the result of database queries.

Query caching requires a [[yii\db\Connection|DB connection]] and a valid `cache` application component. The basic usage of query caching is as follows, assuming `$db` is a [[yii\db\Connection]] instance:

```
$result = $db->cache(function ($db) {

    // the result of the SQL query will be served from the cache
    // if query caching is enabled and the query result is found in the cache
    return $db->createCommand('SELECT * FROM customer WHERE id=1')->queryOne();

});
```

Query caching can be used for DAO as well as ActiveRecord:

```
$result = Customer::getDb()->cache(function ($db) {
    return Customer::find()->where(['id' => 1])->one();
});
```

> Info: Some DBMS (e.g. MySQL) also support query caching on the DB server-side. You may choose to use either query caching mechanism. The query caching described above has the advantage that you may specify flexible cache dependencies and are potentially more efficient.

Cache Flushing

When you need to invalidate all the stored cache data, you can call [[yii\caching\Cache::flush()]].

You can flush the cache from the console by calling `yii cache/flush` as well.

- `yii cache` : lists the available caches in application
- `yii cache/flush cache1 cache2` : flushes the cache components `cache1` , `cache2` (you can pass multiple component names separated with space)
- `yii cache/flush-all` : flushes all cache components in the application

> Info: Console application uses a separate configuration file by default. Ensure, that you have the same caching components in your web and console application configs to reach the proper effect.

Configurations

Query caching has three global configurable options through [[yii\db\Connection]]:

- [[yii\db\Connection::enableQueryCache|enableQueryCache]]: whether to turn on or off query caching. It defaults to `true` . Note that to effectively turn on query caching, you also need to have a valid cache, as specified by [[yii\db\Connection::queryCache|queryCache]].

- [[yii\db\Connection::queryCacheDuration|queryCacheDuration]]: this represents the number of seconds that a query result can remain valid in the cache. You can use 0 to indicate a query result should remain in the cache forever. This property is the default value used when [[yii\db\Connection::cache()]] is called without specifying a duration.
- [[yii\db\Connection::queryCache|queryCache]]: this represents the ID of the cache application component. It defaults to `'cache'`. Query caching is enabled only if there is a valid cache application component.

Usages

You can use [[yii\db\Connection::cache()]] if you have multiple SQL queries that need to take advantage of query caching. The usage is as follows,

```
$duration = 60;     // cache query results for 60 seconds.
$dependency = ...;  // optional dependency

$result = $db->cache(function ($db) {

    // ... perform SQL queries here ...

    return $result;

}, $duration, $dependency);
```

Any SQL queries in the anonymous function will be cached for the specified duration with the specified dependency. If the result of a query is found valid in the cache, the query will be skipped and the result will be served from the cache instead. If you do not specify the `$duration` parameter, the value of [[yii\db\Connection::queryCacheDuration|queryCacheDuration]] will be used instead.

Sometimes within `cache()`, you may want to disable query caching for some particular queries. You can use [[yii\db\Connection::noCache()]] in this case.

```
$result = $db->cache(function ($db) {

    // SQL queries that use query caching

    $db->noCache(function ($db) {

        // SQL queries that do not use query caching

    });

    // ...

    return $result;
});
```

If you just want to use query caching for a single query, you can call [[yii\db\Command::cache()]] when building the command. For example,

```
// use query caching and set query cache duration to be 60 seconds
$customer = $db->createCommand('SELECT * FROM customer WHERE id=1')->cache(60)->queryO
ne();
```

You can also use [[yii\db\Command::noCache()]] to disable query caching for a single command. For example,

```
$result = $db->cache(function ($db) {

    // SQL queries that use query caching

    // do not use query caching for this command
    $customer = $db->createCommand('SELECT * FROM customer WHERE id=1')->noCache()->qu
eryOne();

    // ...

    return $result;
});
```

Limitations

Query caching does not work with query results that contain resource handlers. For example, when using the `BLOB` column type in some DBMS, the query result will return a resource handler for the column data.

Some caching storage has size limitation. For example, memcache limits the maximum size of each entry to be 1MB. Therefore, if the size of a query result exceeds this limit, the caching will fail.

Fragment Caching

Fragment caching refers to caching a fragment of a Web page. For example, if a page displays a summary of yearly sale in a table, you can store this table in cache to eliminate the time needed to generate this table for each request. Fragment caching is built on top of data caching.

To use fragment caching, use the following construct in a view:

```
if ($this->beginCache($id)) {

    // ... generate content here ...

    $this->endCache();
}
```

That is, enclose content generation logic in a pair of [[yii\base\View::beginCache()|beginCache()]] and [[yii\base\View::endCache()|endCache()]] calls. If the content is found in the cache, [[yii\base\View::beginCache()|beginCache()]] will render the cached content and return `false`, thus skip the content generation logic. Otherwise, your content generation logic will be called, and when [[yii\base\View::endCache()|endCache()]] is called, the generated content will be captured and stored in the cache.

Like data caching, a unique `$id` is needed to identify a content cache.

Caching Options

You may specify additional options about fragment caching by passing the option array as the second parameter to the [[yii\base\View::beginCache()|beginCache()]] method. Behind the scene, this option array will be used to configure a [[yii\widgets\FragmentCache]] widget which implements the actual fragment caching functionality.

Duration

Perhaps the most commonly used option of fragment caching is [[yii\widgets\FragmentCache::duration|duration]]. It specifies for how many seconds the content can remain valid in a cache. The following code caches the content fragment for at most one hour:

```
if ($this->beginCache($id, ['duration' => 3600])) {

    // ... generate content here ...

    $this->endCache();
}
```

If the option is not set, it will take the default value 60, which means the cached content will expire in 60 seconds.

Dependencies

Like data caching, content fragment being cached can also have dependencies. For example, the content of a post being displayed depends on whether or not the post is modified.

To specify a dependency, set the [[yii\widgets\FragmentCache::dependency|dependency]] option, which can be either an [[yii\caching\Dependency]] object or a configuration array for creating a dependency object. The following code specifies that the fragment content depends on the change of the `updated_at` column value:

```
$dependency = [
    'class' => 'yii\caching\DbDependency',
    'sql' => 'SELECT MAX(updated_at) FROM post',
];

if ($this->beginCache($id, ['dependency' => $dependency])) {

    // ... generate content here ...

    $this->endCache();
}
```

Variations

Content being cached may be variated according to some parameters. For example, for a Web application supporting multiple languages, the same piece of view code may generate the content in different languages. Therefore, you may want to make the cached content variated according to the current application language.

To specify cache variations, set the [[yii\widgets\FragmentCache::variations|variations]] option, which should be an array of scalar values, each representing a particular variation factor. For example, to make the cached content variated by the language, you may use the following code:

```
if ($this->beginCache($id, ['variations' => [Yii::$app->language]])) {

    // ... generate content here ...

    $this->endCache();
}
```

Toggling Caching

Sometimes you may want to enable fragment caching only when certain conditions are met. For example, for a page displaying a form, you only want to cache the form when it is initially requested (via GET request). Any subsequent display (via POST request) of the form should not be cached because the form may contain user input. To do so, you may set the [[yii\widgets\FragmentCache::enabled|enabled]] option, like the following:

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... generate content here ...

    $this->endCache();
}
```

Nested Caching

Fragment caching can be nested. That is, a cached fragment can be enclosed within another fragment which is also cached. For example, the comments are cached in an inner fragment cache, and they are cached together with the post content in an outer fragment cache. The following code shows how two fragment caches can be nested:

```
if ($this->beginCache($id1)) {

    // ...content generation logic...

    if ($this->beginCache($id2, $options2)) {

        // ...content generation logic...

        $this->endCache();
    }

    // ...content generation logic...

    $this->endCache();
}
```

Different caching options can be set for the nested caches. For example, the inner caches and the outer caches can use different cache duration values. Even when the data cached in the outer cache is invalidated, the inner cache may still provide the valid inner fragment. However, it is not true vice versa. If the outer cache is evaluated to be valid, it will continue to provide the same cached copy even after the content in the inner cache has been invalidated. Therefore, you must be careful in setting the durations or the dependencies of the nested caches, otherwise the outdated inner fragments may be kept in the outer fragment.

Dynamic Content

When using fragment caching, you may encounter the situation where a large fragment of content is relatively static except at one or a few places. For example, a page header may display the main menu bar together with the name of the current user. Another problem is that the content being cached may contain PHP code that must be executed for every request (e.g. the code for registering an asset bundle). Both problems can be solved by the so-called *dynamic content* feature.

A dynamic content means a fragment of output that should not be cached even if it is enclosed within a fragment cache. To make the content dynamic all the time, it has to be generated by executing some PHP code for every request, even if the enclosing content is being served from cache.

You may call [[yii\base\View::renderDynamic()]] within a cached fragment to insert dynamic content at the desired place, like the following,

```
if ($this->beginCache($id1)) {

    // ...content generation logic...

    echo $this->renderDynamic('return Yii::$app->user->identity->name;');

    // ...content generation logic...

    $this->endCache();
}
```

The [[yii\base\View::renderDynamic()|renderDynamic()]] method takes a piece of PHP code as its parameter. The return value of the PHP code is treated as the dynamic content. The same PHP code will be executed for every request, no matter the enclosing fragment is being served from cached or not.

Page Caching

Page caching refers to caching the content of a whole page on the server-side. Later when the same page is requested again, its content will be served from the cache instead of regenerating it from scratch.

Page caching is supported by [[yii\filters\PageCache]], an action filter. It can be used like the following in a controller class:

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\PageCache',
            'only' => ['index'],
            'duration' => 60,
            'variations' => [
                \Yii::$app->language,
            ],
            'dependency' => [
                'class' => 'yii\caching\DbDependency',
                'sql' => 'SELECT COUNT(*) FROM post',
            ],
        ],
    ];
}
```

The above code states that page caching should be used only for the `index` action. The page content should be cached for at most 60 seconds and should be variated by the current application language and the cached page should be invalidated if the total number of posts is changed.

As you can see, page caching is very similar to fragment caching. They both support options such as `duration`, `dependencies`, `variations`, and `enabled`. Their main difference is that page caching is implemented as an action filter while fragment caching a widget.

You can use fragment caching as well as dynamic content together with page caching.

HTTP Caching

Besides server-side caching that we have described in the previous sections, Web applications may also exploit client-side caching to save the time for generating and transmitting the same page content.

To use client-side caching, you may configure [[yii\filters\HttpCache]] as a filter for controller actions whose rendering result may be cached on the client-side. [[yii\filters\HttpCache|HttpCache]] only works for `GET` and `HEAD` requests. It can handle three kinds of cache-related HTTP headers for these requests:

- [[yii\filters\HttpCache::lastModified|Last-Modified]]
- [[yii\filters\HttpCache::etagSeed|Etag]]
- [[yii\filters\HttpCache::cacheControlHeader|Cache-Control]]

`Last-Modified` Header

The `Last-Modified` header uses a timestamp to indicate if the page has been modified since the client caches it.

You may configure the [[yii\filters\HttpCache::lastModified]] property to enable sending the `Last-Modified` header. The property should be a PHP callable returning a UNIX timestamp about the page modification time. The signature of the PHP callable should be as follows,

```
/**
 * @param Action $action the action object that is being handled currently
 * @param array $params the value of the "params" property
 * @return int a UNIX timestamp representing the page modification time
 */
function ($action, $params)
```

The following is an example of making use of the `Last-Modified` header:

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('post')->max('updated_at');
            },
        ],
    ];
}
```

The above code states that HTTP caching should be enabled for the `index` action only. It should generate a `Last-Modified` HTTP header based on the last update time of posts. When a browser visits the `index` page for the first time, the page will be generated on the server and sent to the browser; If the browser visits the same page again and there is no post being modified during the period, the server will not re-generate the page, and the browser will use the cached version on the client-side. As a result, server-side rendering and page content transmission are both skipped.

`ETag` Header

The "Entity Tag" (or `ETag` for short) header use a hash to represent the content of a page. If the page is changed, the hash will be changed as well. By comparing the hash kept on the client-side with the hash generated on the server-side, the cache may determine whether the page has been changed and should be re-transmitted.

You may configure the [[yii\filters\HttpCache::etagSeed]] property to enable sending the `ETag` header. The property should be a PHP callable returning a seed for generating the ETag hash. The signature of the PHP callable should be as follows,

```
/**
 * @param Action $action the action object that is being handled currently
 * @param array $params the value of the "params" property
 * @return string a string used as the seed for generating an ETag hash
 */
function ($action, $params)
```

The following is an example of making use of the `ETag` header:

```
public function behaviors()
{
    return [
        [
            'class' => 'yii\filters\HttpCache',
            'only' => ['view'],
            'etagSeed' => function ($action, $params) {
                $post = $this->findModel(\Yii::$app->request->get('id'));
                return serialize([$post->title, $post->content]);
            },
        ],
    ];
}
```

The above code states that HTTP caching should be enabled for the `view` action only. It should generate an `ETag` HTTP header based on the title and content of the requested post. When a browser visits the `view` page for the first time, the page will be generated on the server and sent to the browser; If the browser visits the same page again and there is no change to the title and content of the post, the server will not re-generate the page, and the browser will use the cached version on the client-side. As a result, server-side rendering and page content transmission are both skipped.

ETags allow more complex and/or more precise caching strategies than `Last-Modified` headers. For instance, an ETag can be invalidated if the site has switched to another theme.

Expensive ETag generation may defeat the purpose of using `HttpCache` and introduce unnecessary overhead, since they need to be re-evaluated on every request. Try to find a simple expression that invalidates the cache if the page content has been modified.

> Note: In compliance to RFC 7232, `HttpCache` will send out both `ETag` and `Last-Modified` headers if they are both configured. And if the client sends both of the `If-None-Match` header and the `If-Modified-Since` header, only the former will be respected.

`Cache-Control` Header

The `Cache-Control` header specifies the general caching policy for pages. You may send it by configuring the [[yii\filters\HttpCache::cacheControlHeader]] property with the header value. By default, the following header will be sent:

```
Cache-Control: public, max-age=3600
```

Session Cache Limiter

When a page uses session, PHP will automatically send some cache-related HTTP headers as specified in the `session.cache_limiter` PHP INI setting. These headers may interfere or disable the caching that you want from `HttpCache`. To prevent this problem, by default `HttpCache` will disable sending these headers automatically. If you want to change this behavior, you should configure the [[yii\filters\HttpCache::sessionCacheLimiter]] property. The property can take a string value, including `public`, `private`, `private_no_expire`, and `nocache`. Please refer to the PHP manual about session_cache_limiter() for explanations about these values.

SEO Implications

Search engine bots tend to respect cache headers. Since some crawlers have a limit on how many pages per domain they process within a certain time span, introducing caching headers may help indexing your site as they reduce the number of pages that need to be processed.

Quick Start

Yii provides a whole set of tools to simplify the task of implementing RESTful Web Service APIs. In particular, Yii supports the following features about RESTful APIs:

- Quick prototyping with support for common APIs for Active Record;
- Response format negotiation (supporting JSON and XML by default);
- Customizable object serialization with support for selectable output fields;
- Proper formatting of collection data and validation errors;
- Support for HATEOAS;
- Efficient routing with proper HTTP verb check;
- Built-in support for the `OPTIONS` and `HEAD` verbs;
- Authentication and authorization;
- Data caching and HTTP caching;
- Rate limiting;

In the following, we use an example to illustrate how you can build a set of RESTful APIs with some minimal coding effort.

Assume you want to expose the user data via RESTful APIs. The user data are stored in the `user` DB table, and you have already created the active record class `app\models\User` to access the user data.

Creating a Controller

First, create a controller class `app\controllers\UserController` as follows:

```
namespace app\controllers;

use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
}
```

The controller class extends from [[yii\rest\ActiveController]], which implements a common set of RESTful actions. By specifying [[yii\rest\ActiveController::modelClass|modelClass]] as `app\models\User`, the controller knows which model can be used for fetching and manipulating data.

Configuring URL Rules

Then, modify the configuration of the `urlManager` component in your application configuration:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

The above configuration mainly adds a URL rule for the `user` controller so that the user data can be accessed and manipulated with pretty URLs and meaningful HTTP verbs.

> Note: Yii will automatically pluralize controller names for use in endpoints (see Trying it Out section below). You can configure this using the [[yii\rest\UrlRule::$pluralize]] property.

Enabling JSON Input

To let the API accept input data in JSON format, configure the [[yii\web\Request::$parsers|parsers]] property of the `request` application component to use the [[yii\web\JsonParser]] for JSON input:

```
'request' => [
    'parsers' => [
        'application/json' => 'yii\web\JsonParser',
    ]
]
```

> Info: The above configuration is optional. Without the above configuration, the API would only recognize `application/x-www-form-urlencoded` and `multipart/form-data` input formats.

Trying it Out

With the above minimal amount of effort, you have already finished your task of creating the RESTful APIs for accessing the user data. The APIs you have created include:

- `GET /users` : list all users page by page;
- `HEAD /users` : show the overview information of user listing;
- `POST /users` : create a new user;
- `GET /users/123` : return the details of the user 123;
- `HEAD /users/123` : show the overview information of user 123;
- `PATCH /users/123` and `PUT /users/123` : update the user 123;
- `DELETE /users/123` : delete the user 123;
- `OPTIONS /users` : show the supported verbs regarding endpoint `/users` ;
- `OPTIONS /users/123` : show the supported verbs regarding endpoint `/users/123` .

You may access your APIs with the `curl` command like the following,

```
$ curl -i -H "Accept:application/json" "http://localhost/users"

HTTP/1.1 200 OK
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

[
    {
        "id": 1,
        ...
    },
    {
        "id": 2,
        ...
    },
    ...
]
```

Try changing the acceptable content type to be `application/xml` , and you will see the result is returned in XML format:

```
$ curl -i -H "Accept:application/xml" "http://localhost/users"

HTTP/1.1 200 OK
...
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/xml

<?xml version="1.0" encoding="UTF-8"?>
<response>
    <item>
        <id>1</id>
        ...
    </item>
    <item>
        <id>2</id>
        ...
    </item>
    ...
</response>
```

The following command will create a new user by sending a POST request with the user data in JSON format:

```
$ curl -i -H "Accept:application/json" -H "Content-Type:application/json" -XPOST "http
://localhost/users" -d '{"username": "example", "email": "user@example.com"}'

HTTP/1.1 201 Created
...
Location: http://localhost/users/1
Content-Length: 99
Content-Type: application/json; charset=UTF-8

{"id":1,"username":"example","email":"user@example.com","created_at":1414674789,"updat
ed_at":1414674789}
```

> Tip: You may also access your APIs via Web browser by entering the URL `http://localhost/users`. However, you may need some browser plugins to send specific request headers.

As you can see, in the response headers, there is information about the total count, page count, etc. There are also links that allow you to navigate to other pages of data. For example, `http://localhost/users?page=2` would give you the next page of the user data.

Using the `fields` and `expand` parameters, you may also specify which fields should be included in the result. For example, the URL `http://localhost/users?fields=id,email` will only return the `id` and `email` fields.

> Info: You may have noticed that the result of `http://localhost/users` includes some sensitive fields, such as `password_hash`, `auth_key`. You certainly do not want these to appear in your API result. You can and should filter out these fields as described in the Resources section.

Summary

Using the Yii RESTful API framework, you implement an API endpoint in terms of a controller action, and you use a controller to organize the actions that implement the endpoints for a single type of resource.

Resources are represented as data models which extend from the [[yii\base\Model]] class. If you are working with databases (relational or NoSQL), it is recommended you use [[yii\db\ActiveRecord|ActiveRecord]] to represent resources.

You may use [[yii\rest\UrlRule]] to simplify the routing to your API endpoints.

While not required, it is recommended that you develop your RESTful APIs as a separate application, different from your Web front end and back end for easier maintenance.

Resources

RESTful APIs are all about accessing and manipulating *resources*. You may view resources as models in the MVC paradigm.

While there is no restriction in how to represent a resource, in Yii you usually would represent resources in terms of objects of [[yii\base\Model]] or its child classes (e.g. [[yii\db\ActiveRecord]]), for the following reasons:

- [[yii\base\Model]] implements the [[yii\base\Arrayable]] interface, which allows you to customize how you want to expose resource data through RESTful APIs.
- [[yii\base\Model]] supports input validation, which is useful if your RESTful APIs need to support data input.
- [[yii\db\ActiveRecord]] provides powerful DB data access and manipulation support, which makes it a perfect fit if your resource data is stored in databases.

In this section, we will mainly describe how a resource class extending from [[yii\base\Model]] (or its child classes) can specify what data may be returned via RESTful APIs. If the resource class does not extend from [[yii\base\Model]], then all its public member variables will be returned.

Fields

When including a resource in a RESTful API response, the resource needs to be serialized into a string. Yii breaks this process into two steps. First, the resource is converted into an array by [[yii\rest\Serializer]]. Second, the array is serialized into a string in a requested format (e.g. JSON, XML) by [[yii\web\ResponseFormatterInterface|response formatters]]. The first step is what you should mainly focus when developing a resource class.

By overriding [[yii\base\Model::fields()|fields()]] and/or [[yii\base\Model::extraFields()|extraFields()]], you may specify what data, called *fields*, in the resource can be put into its array representation. The difference between these two methods is that the former specifies the default set of fields which should be included in the array representation, while the latter specifies additional fields which may be included in the array if an end user requests for them via the `expand` query parameter. For example,

```
// returns all fields as declared in fields()
http://localhost/users

// only returns field id and email, provided they are declared in fields()
http://localhost/users?fields=id,email

// returns all fields in fields() and field profile if it is in extraFields()
http://localhost/users?expand=profile

// only returns field id, email and profile, provided they are in fields() and extraFi
elds()
http://localhost/users?fields=id,email&expand=profile
```

Overriding `fields()`

By default, [[yii\base\Model::fields()]] returns all model attributes as fields, while [[yii\db\ActiveRecord::fields()]] only returns the attributes which have been populated from DB.

You can override `fields()` to add, remove, rename or redefine fields. The return value of `fields()` should be an array. The array keys are the field names, and the array values are the corresponding field definitions which can be either property/attribute names or anonymous functions returning the corresponding field values. In the special case when a field name is the same as its defining attribute name, you can omit the array key. For example,

```
// explicitly list every field, best used when you want to make sure the changes
// in your DB table or model attributes do not cause your field changes (to keep API b
ackward compatibility).
public function fields()
{
    return [
        // field name is the same as the attribute name
        'id',
        // field name is "email", the corresponding attribute name is "email_address"
        'email' => 'email_address',
        // field name is "name", its value is defined by a PHP callback
        'name' => function ($model) {
            return $model->first_name . ' ' . $model->last_name;
        },
    ];
}

// filter out some fields, best used when you want to inherit the parent implementation

// and blacklist some sensitive fields.
public function fields()
{
    $fields = parent::fields();

    // remove fields that contain sensitive information
    unset($fields['auth_key'], $fields['password_hash'], $fields['password_reset_token'
]);

    return $fields;
}
```

> Warning: Because by default all attributes of a model will be included in the API result, you should examine your data to make sure they do not contain sensitive information. If there is such information, you should override `fields()` to filter them out. In the above example, we choose to filter out `auth_key`, `password_hash` and `password_reset_token`.

Overriding `extraFields()`

By default, [[yii\base\Model::extraFields()]] returns an empty array, while [[yii\db\ActiveRecord::extraFields()]] returns the names of the relations that have been populated from DB.

The return data format of `extraFields()` is the same as that of `fields()`. Usually, `extraFields()` is mainly used to specify fields whose values are objects. For example, given the following field declaration,

```
public function fields()
{
    return ['id', 'email'];
}

public function extraFields()
{
    return ['profile'];
}
```

the request with `http://localhost/users?fields=id,email&expand=profile` may return the following JSON data:

```
[
    {
        "id": 100,
        "email": "100@example.com",
        "profile": {
            "id": 100,
            "age": 30,
        }
    },
    ...
]
```

Links

HATEOAS, an abbreviation for Hypermedia as the Engine of Application State, promotes that RESTful APIs should return information that allows clients to discover actions supported for the returned resources. The key of HATEOAS is to return a set of hyperlinks with relation information when resource data are served by the APIs.

Your resource classes may support HATEOAS by implementing the [[yii\web\Linkable]] interface. The interface contains a single method [[yii\web\Linkable::getLinks()|getLinks()]] which should return a list of [[yii\web\Link|links]]. Typically, you should return at least the `self` link representing the URL to the resource object itself. For example,

```
use yii\base\Model;
use yii\web\Link; // represents a link object as defined in JSON Hypermedia API Langua
ge.
use yii\web\Linkable;
use yii\helpers\Url;

class UserResource extends Model implements Linkable
{
    public $id;
    public $email;

    //...

    public function fields()
    {
        return ['id', 'email'];
    }

    public function extraFields()
    {
        return ['profile'];
    }

    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user/view', 'id' => $this->id], true),
            'edit' => Url::to(['user/view', 'id' => $this->id], true),
            'profile' => Url::to(['user/profile/view', 'id' => $this->id], true),
            'index' => Url::to(['users'], true),
        ];
    }
}
```

When a `UserResource` object is returned in a response, it will contain a `_links` element representing the links related to the user, for example,

```
{
    "id": 100,
    "email": "user@example.com",
    // ...
    "_links" => {
        "self": {
            "href": "https://example.com/users/100"
        },
        "edit": {
            "href": "https://example.com/users/100"
        },
        "profile": {
            "href": "https://example.com/users/profile/100"
        },
        "index": {
            "href": "https://example.com/users"
        }
    }
}
```

Collections

Resource objects can be grouped into *collections*. Each collection contains a list of resource objects of the same type.

While collections can be represented as arrays, it is usually more desirable to represent them as data providers. This is because data providers support sorting and pagination of resources, which is a commonly needed feature for RESTful APIs returning collections. For example, the following action returns a data provider about the post resources:

```php
namespace app\controllers;

use yii\rest\Controller;
use yii\data\ActiveDataProvider;
use app\models\Post;

class PostController extends Controller
{
    public function actionIndex()
    {
        return new ActiveDataProvider([
            'query' => Post::find(),
        ]);
    }
}
```

When a data provider is being sent in a RESTful API response, [[yii\rest\Serializer]] will take out the current page of resources and serialize them as an array of resource objects. Additionally, [[yii\rest\Serializer]] will also include the pagination information by the following HTTP headers:

- `X-Pagination-Total-Count` : The total number of resources;
- `X-Pagination-Page-Count` : The number of pages;
- `X-Pagination-Current-Page` : The current page (1-based);
- `X-Pagination-Per-Page` : The number of resources in each page;
- `Link` : A set of navigational links allowing client to traverse the resources page by page.

An example may be found in the Quick Start section.

Controllers

After creating the resource classes and specifying how resource data should be formatted, the next thing to do is to create controller actions to expose the resources to end users through RESTful APIs.

Yii provides two base controller classes to simplify your work of creating RESTful actions: [[yii\rest\Controller]] and [[yii\rest\ActiveController]]. The difference between these two controllers is that the latter provides a default set of actions that are specifically designed to deal with resources represented as Active Record. So if you are using Active Record and are comfortable with the provided built-in actions, you may consider extending your controller classes from [[yii\rest\ActiveController]], which will allow you to create powerful RESTful APIs with minimal code.

Both [[yii\rest\Controller]] and [[yii\rest\ActiveController]] provide the following features, some of which will be described in detail in the next few sections:

- HTTP method validation;
- Content negotiation and Data formatting;
- Authentication;
- Rate limiting.

[[yii\rest\ActiveController]] in addition provides the following features:

- A set of commonly needed actions: `index`, `view`, `create`, `update`, `delete`, `options`;
- User authorization in regard to the requested action and resource.

Creating Controller Classes

When creating a new controller class, a convention in naming the controller class is to use the type name of the resource and use singular form. For example, to serve user information, the controller may be named as `UserController`.

Creating a new action is similar to creating an action for a Web application. The only difference is that instead of rendering the result using a view by calling the `render()` method, for RESTful actions you directly return the data. The [[yii\rest\Controller::serializer|serializer]] and the [[yii\web\Response|response object]] will handle the conversion from the original data to the requested format. For example,

```
public function actionView($id)
{
    return User::findOne($id);
}
```

Filters

Most RESTful API features provided by [[yii\rest\Controller]] are implemented in terms of filters. In particular, the following filters will be executed in the order they are listed:

- [[yii\filters\ContentNegotiator|contentNegotiator]]: supports content negotiation, to be explained in the Response Formatting section;
- [[yii\filters\VerbFilter|verbFilter]]: supports HTTP method validation;
- [[yii\filters\auth\AuthMethod|authenticator]]: supports user authentication, to be explained in the Authentication section;
- [[yii\filters\RateLimiter|rateLimiter]]: supports rate limiting, to be explained in the Rate Limiting section.

These named filters are declared in the [[yii\rest\Controller::behaviors()|behaviors()]] method. You may override this method to configure individual filters, disable some of them, or add your own filters. For example, if you only want to use HTTP basic authentication, you may write the following code:

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::className(),
    ];
    return $behaviors;
}
```

CORS

Adding the Cross-Origin Resource Sharing filter to a controller is a bit more complicated than adding other filters described above, because the CORS filter has to be applied before authentication methods and thus needs a slightly different approach compared to other filters. Also authentication has to be disabled for the CORS Preflight requests so that a browser can safely determine whether a request can be made beforehand without the need

for sending authentication credentials. The following shows the code that is needed to add the [[yii\filters\Cors]] filter to an existing controller that extends from [[yii\rest\ActiveController]]:

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();

    // remove authentication filter
    $auth = $behaviors['authenticator'];
    unset($behaviors['authenticator']);

    // add CORS filter
    $behaviors['corsFilter'] = [
        'class' => \yii\filters\Cors::className(),
    ];

    // re-add authentication filter
    $behaviors['authenticator'] = $auth;
    // avoid authentication on CORS-pre-flight requests (HTTP OPTIONS method)
    $behaviors['authenticator']['except'] = ['options'];

    return $behaviors;
}
```

Extending `ActiveController`

If your controller class extends from [[yii\rest\ActiveController]], you should set its [[yii\rest\ActiveController::modelClass|modelClass]] property to be the name of the resource class that you plan to serve through this controller. The class must extend from [[yii\db\ActiveRecord]].

Customizing Actions

By default, [[yii\rest\ActiveController]] provides the following actions:

- [[yii\rest\IndexAction|index]]: list resources page by page;
- [[yii\rest\ViewAction|view]]: return the details of a specified resource;
- [[yii\rest\CreateAction|create]]: create a new resource;
- [[yii\rest\UpdateAction|update]]: update an existing resource;
- [[yii\rest\DeleteAction|delete]]: delete the specified resource;
- [[yii\rest\OptionsAction|options]]: return the supported HTTP methods.

All these actions are declared through the [[yii\rest\ActiveController::actions()|actions()]] method. You may configure these actions or disable some of them by overriding the `actions()` method, like shown the following,

```
public function actions()
{
    $actions = parent::actions();

    // disable the "delete" and "create" actions
    unset($actions['delete'], $actions['create']);

    // customize the data provider preparation with the "prepareDataProvider()" method
    $actions['index']['prepareDataProvider'] = [$this, 'prepareDataProvider'];

    return $actions;
}

public function prepareDataProvider()
{
    // prepare and return a data provider for the "index" action
}
```

Please refer to the class references for individual action classes to learn what configuration options are available.

Performing Access Check

When exposing resources through RESTful APIs, you often need to check if the current user has the permission to access and manipulate the requested resource(s). With [[yii\rest\ActiveController]], this can be done by overriding the [[yii\rest\ActiveController::checkAccess()|checkAccess()]] method like the following,

```
/**
 * Checks the privilege of the current user.
 *
 * This method should be overridden to check whether the current user has the privileg
e
 * to run the specified action against the specified data model.
 * If the user does not have access, a [[ForbiddenHttpException]] should be thrown.
 *
 * @param string $action the ID of the action to be executed
 * @param \yii\base\Model $model the model to be accessed. If `null`, it means no spec
ific model is being accessed.
 * @param array $params additional parameters
 * @throws ForbiddenHttpException if the user does not have access
 */
public function checkAccess($action, $model = null, $params = [])
{
    // check if the user can access $action and $model
    // throw ForbiddenHttpException if access should be denied
    if ($action === 'update' || $action === 'delete') {
        if ($model->author_id !== \Yii::$app->user->id)
            throw new \yii\web\ForbiddenHttpException(sprintf('You can only %s article
s that you\'ve created.', $action));
    }
}
```

The `checkAccess()` method will be called by the default actions of [[yii\rest\ActiveController]]. If you create new actions and also want to perform access check, you should call this method explicitly in the new actions.

> Tip: You may implement `checkAccess()` by using the Role-Based Access Control (RBAC) component.

Routing

With resource and controller classes ready, you can access the resources using the URL like `http://localhost/index.php?r=user/create`, similar to what you can do with normal Web applications.

In practice, you usually want to enable pretty URLs and take advantage of HTTP verbs. For example, a request `POST /users` would mean accessing the `user/create` action. This can be done easily by configuring the `urlManager` application component in the application configuration like the following:

```
'urlManager' => [
    'enablePrettyUrl' => true,
    'enableStrictParsing' => true,
    'showScriptName' => false,
    'rules' => [
        ['class' => 'yii\rest\UrlRule', 'controller' => 'user'],
    ],
]
```

Compared to the URL management for Web applications, the main new thing above is the use of [[yii\rest\UrlRule]] for routing RESTful API requests. This special URL rule class will create a whole set of child URL rules to support routing and URL creation for the specified controller(s). For example, the above code is roughly equivalent to the following rules:

```
[
    'PUT,PATCH users/<id>' => 'user/update',
    'DELETE users/<id>' => 'user/delete',
    'GET,HEAD users/<id>' => 'user/view',
    'POST users' => 'user/create',
    'GET,HEAD users' => 'user/index',
    'users/<id>' => 'user/options',
    'users' => 'user/options',
]
```

And the following API endpoints are supported by this rule:

- `GET /users`: list all users page by page;
- `HEAD /users`: show the overview information of user listing;
- `POST /users`: create a new user;
- `GET /users/123`: return the details of the user 123;
- `HEAD /users/123`: show the overview information of user 123;

- `PATCH /users/123` and `PUT /users/123` : update the user 123;
- `DELETE /users/123` : delete the user 123;
- `OPTIONS /users` : show the supported verbs regarding endpoint `/users` ;
- `OPTIONS /users/123` : show the supported verbs regarding endpoint `/users/123` .

You may configure the `only` and `except` options to explicitly list which actions to support or which actions should be disabled, respectively. For example,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'except' => ['delete', 'create', 'update'],
],
```

You may also configure `patterns` or `extraPatterns` to redefine existing patterns or add new patterns supported by this rule. For example, to support a new action `search` by the endpoint `GET /users/search` , configure the `extraPatterns` option as follows,

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => 'user',
    'extraPatterns' => [
        'GET search' => 'search',
    ],
]
```

You may have noticed that the controller ID `user` appears in plural form as `users` in the endpoint URLs. This is because [[yii\rest\UrlRule]] automatically pluralizes controller IDs when creating child URL rules. You may disable this behavior by setting [[yii\rest\UrlRule::pluralize]] to be `false` .

> Info: The pluralization of controller IDs is done by [[yii\helpers\Inflector::pluralize()]]. The method respects special pluralization rules. For example, the word `box` will be pluralized as `boxes` instead of `boxs` .

In case when the automatic pluralization does not meet your requirement, you may also configure the [[yii\rest\UrlRule::controller]] property to explicitly specify how to map a name used in endpoint URLs to a controller ID. For example, the following code maps the name `u` to the controller ID `user` .

```
[
    'class' => 'yii\rest\UrlRule',
    'controller' => ['u' => 'user'],
]
```

Response Formatting

When handling a RESTful API request, an application usually takes the following steps that are related with response formatting:

1. Determine various factors that may affect the response format, such as media type, language, version, etc. This process is also known as content negotiation.
2. Convert resource objects into arrays, as described in the Resources section. This is done by [[yii\rest\Serializer]].
3. Convert arrays into a string in the format as determined by the content negotiation step. This is done by [[yii\web\ResponseFormatterInterface|response formatters]] registered with the [[yii\web\Response::formatters|formatters]] property of the `response` application component.

Content Negotiation

Yii supports content negotiation via the [[yii\filters\ContentNegotiator]] filter. The RESTful API base controller class [[yii\rest\Controller]] is equipped with this filter under the name of `contentNegotiator`. The filter provides response format negotiation as well as language negotiation. For example, if a RESTful API request contains the following header,

```
Accept: application/json; q=1.0, */*; q=0.1
```

it will get a response in JSON format, like the following:

```
$ curl -i -H "Accept: application/json; q=1.0, */*; q=0.1" "http://localhost/users"

HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

[
    {
        "id": 1,
        ...
    },
    {
        "id": 2,
        ...
    },
    ...
]
```

Behind the scene, before a RESTful API controller action is executed, the [[yii\filters\ContentNegotiator]] filter will check the `Accept` HTTP header in the request and set the [[yii\web\Response::format|response format]] to be `'json'`. After the action is executed and returns the resulting resource object or collection, [[yii\rest\Serializer]] will convert the result into an array. And finally, [[yii\web\JsonResponseFormatter]] will serialize the array into a JSON string and include it in the response body.

By default, RESTful APIs support both JSON and XML formats. To support a new format, you should configure the [[yii\filters\ContentNegotiator::formats|formats]] property of the `contentNegotiator` filter like the following in your API controller classes:

```php
use yii\web\Response;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['contentNegotiator']['formats']['text/html'] = Response::FORMAT_HTML;
    return $behaviors;
}
```

The keys of the `formats` property are the supported MIME types, while the values are the corresponding response format names which must be supported in [[yii\web\Response::formatters]].

Data Serializing

As we have described above, [[yii\rest\Serializer]] is the central piece responsible for converting resource objects or collections into arrays. It recognizes objects implementing [[yii\base\Arrayable]] as well as [[yii\data\DataProviderInterface]]. The former is mainly implemented by resource objects, while the latter resource collections.

You may configure the serializer by setting the [[yii\rest\Controller::serializer]] property with a configuration array. For example, sometimes you may want to help simplify the client development work by including pagination information directly in the response body. To do so, configure the [[yii\rest\Serializer::collectionEnvelope]] property as follows:

```
use yii\rest\ActiveController;

class UserController extends ActiveController
{
    public $modelClass = 'app\models\User';
    public $serializer = [
        'class' => 'yii\rest\Serializer',
        'collectionEnvelope' => 'items',
    ];
}
```

You may then get the following response for request `http://localhost/users`:

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
X-Powered-By: PHP/5.4.20
X-Pagination-Total-Count: 1000
X-Pagination-Page-Count: 50
X-Pagination-Current-Page: 1
X-Pagination-Per-Page: 20
Link: <http://localhost/users?page=1>; rel=self,
      <http://localhost/users?page=2>; rel=next,
      <http://localhost/users?page=50>; rel=last
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "items": [
        {
            "id": 1,
            ...
        },
        {
            "id": 2,
            ...
        },
        ...
    ],
    "_links": {
        "self": {
            "href": "http://localhost/users?page=1"
        },
        "next": {
            "href": "http://localhost/users?page=2"
        },
        "last": {
            "href": "http://localhost/users?page=50"
        }
    },
    "_meta": {
        "totalCount": 1000,
        "pageCount": 50,
        "currentPage": 1,
        "perPage": 20
    }
}
```

Controlling JSON output

The JSON response is generated by the [[yii\web\JsonResponseFormatter|JsonResponseFormatter]] class which will use the [[yii\helpers\Json|JSON helper]] internally. This formatter can be configured with different

options like for example the [[yii\web\JsonResponseFormatter::$prettyPrint|$prettyPrint]] option, which is useful on development for better readable responses, or [[yii\web\JsonResponseFormatter::$encodeOptions|$encodeOptions]] to control the output of the JSON encoding.

The formatter can be configured in the [[yii\web\Response::formatters|formatters]] property of the `response` application component in the application configuration like the following:

```
'response' => [
    // ...
    'formatters' => [
        \yii\web\Response::FORMAT_JSON => [
            'class' => 'yii\web\JsonResponseFormatter',
            'prettyPrint' => YII_DEBUG, // use "pretty" output in debug mode
            'encodeOptions' => JSON_UNESCAPED_SLASHES | JSON_UNESCAPED_UNICODE,
            // ...
        ],
    ],
],
```

When returning data from a database using the DAO database layer all data will be represented as strings, which is not always the expected result especially numeric values should be represented as numbers in JSON. When using the ActiveRecord layer for retrieving data from the database, the values for numeric columns will be converted to integers when data is fetched from the database in [[yii\db\ActiveRecord::populateRecord()]].

Authentication

Unlike Web applications, RESTful APIs are usually stateless, which means sessions or cookies should not be used. Therefore, each request should come with some sort of authentication credentials because the user authentication status may not be maintained by sessions or cookies. A common practice is to send a secret access token with each request to authenticate the user. Since an access token can be used to uniquely identify and authenticate a user, **API requests should always be sent via HTTPS to prevent man-in-the-middle (MitM) attacks**.

There are different ways to send an access token:

- HTTP Basic Auth: the access token is sent as the username. This should only be used when an access token can be safely stored on the API consumer side. For example, the API consumer is a program running on a server.
- Query parameter: the access token is sent as a query parameter in the API URL, e.g., `https://example.com/users?access-token=xxxxxxxx`. Because most Web servers will keep query parameters in server logs, this approach should be mainly used to serve `JSONP` requests which cannot use HTTP headers to send access tokens.
- OAuth 2: the access token is obtained by the consumer from an authorization server and sent to the API server via HTTP Bearer Tokens, according to the OAuth2 protocol.

Yii supports all of the above authentication methods. You can also easily create new authentication methods.

To enable authentication for your APIs, do the following steps:

1. Configure the `user` application component:
   - Set the [[yii\web\User::enableSession|enableSession]] property to be `false`.
   - Set the [[yii\web\User::loginUrl|loginUrl]] property to be `null` to show a HTTP 403 error instead of redirecting to the login page.
2. Specify which authentication methods you plan to use by configuring the `authenticator` behavior in your REST controller classes.
3. Implement [[yii\web\IdentityInterface::findIdentityByAccessToken()]] in your [[yii\web\User::identityClass|user identity class]].

Step 1 is not required but is recommended for RESTful APIs which should be stateless. When [[yii\web\User::enableSession|enableSession]] is `false`, the user authentication status will NOT be persisted across requests using sessions. Instead, authentication will be performed for every request, which is accomplished by Step 2 and 3.

> Tip: You may configure [[yii\web\User::enableSession|enableSession]] of the `user` application component in application configurations if you are developing RESTful APIs in terms of an application. If you develop RESTful APIs as a module, you may put the following line in the module's `init()` method, like the following:
>
> ```
> public function init()
> {
>     parent::init();
>     \Yii::$app->user->enableSession = false;
> }
> ```

For example, to use HTTP Basic Auth, you may configure the `authenticator` behavior as follows,

```
use yii\filters\auth\HttpBasicAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => HttpBasicAuth::className(),
    ];
    return $behaviors;
}
```

If you want to support all three authentication methods explained above, you can use `CompositeAuth` like the following,

```
use yii\filters\auth\CompositeAuth;
use yii\filters\auth\HttpBasicAuth;
use yii\filters\auth\HttpBearerAuth;
use yii\filters\auth\QueryParamAuth;

public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['authenticator'] = [
        'class' => CompositeAuth::className(),
        'authMethods' => [
            HttpBasicAuth::className(),
            HttpBearerAuth::className(),
            QueryParamAuth::className(),
        ],
    ];
    return $behaviors;
}
```

Each element in `authMethods` should be an auth method class name or a configuration array.

Implementation of `findIdentityByAccessToken()` is application specific. For example, in simple scenarios when each user can only have one access token, you may store the access token in an `access_token` column in the user table. The method can then be readily implemented in the `User` class as follows,

```
use yii\db\ActiveRecord;
use yii\web\IdentityInterface;

class User extends ActiveRecord implements IdentityInterface
{
    public static function findIdentityByAccessToken($token, $type = null)
    {
        return static::findOne(['access_token' => $token]);
    }
}
```

After authentication is enabled as described above, for every API request, the requested controller will try to authenticate the user in its `beforeAction()` step.

If authentication succeeds, the controller will perform other checks (such as rate limiting, authorization) and then run the action. The authenticated user identity information can be retrieved via `Yii::$app->user->identity`.

If authentication fails, a response with HTTP status 401 will be sent back together with other appropriate headers (such as a `WWW-Authenticate` header for HTTP Basic Auth).

Authorization

After a user is authenticated, you probably want to check if he or she has the permission to perform the requested action for the requested resource. This process is called *authorization* which is covered in detail in the Authorization section.

If your controllers extend from [[yii\rest\ActiveController]], you may override the [[yii\rest\ActiveController::checkAccess()|checkAccess()]] method to perform authorization check. The method will be called by the built-in actions provided by [[yii\rest\ActiveController]].

Rate Limiting

To prevent abuse, you should consider adding *rate limiting* to your APIs. For example, you may want to limit the API usage of each user to be at most 100 API calls within a period of 10 minutes. If too many requests are received from a user within the stated period of the time, a response with status code 429 (meaning "Too Many Requests") should be returned.

To enable rate limiting, the [[yii\web\User::identityClass|user identity class]] should implement [[yii\filters\RateLimitInterface]]. This interface requires implementation of three methods:

- `getRateLimit()`: returns the maximum number of allowed requests and the time period (e.g., `[100, 600]` means there can be at most 100 API calls within 600 seconds).
- `loadAllowance()`: returns the number of remaining requests allowed and the corresponding UNIX timestamp when the rate limit was last checked.
- `saveAllowance()`: saves both the number of remaining requests allowed and the current UNIX timestamp.

You may want to use two columns in the user table to record the allowance and timestamp information. With those defined, then `loadAllowance()` and `saveAllowance()` can be implemented to read and save the values of the two columns corresponding to the current authenticated user. To improve performance, you may also consider storing these pieces of information in a cache or NoSQL storage.

Implementation in the `User` model could look like the following:

```
public function getRateLimit($request, $action)
{
    return [$this->rateLimit, 1]; // $rateLimit requests per second
}

public function loadAllowance($request, $action)
{
    return [$this->allowance, $this->allowance_updated_at];
}

public function saveAllowance($request, $action, $allowance, $timestamp)
{
    $this->allowance = $allowance;
    $this->allowance_updated_at = $timestamp;
    $this->save();
}
```

Once the identity class implements the required interface, Yii will automatically use [[yii\filters\RateLimiter]] configured as an action filter for [[yii\rest\Controller]] to perform rate limiting check. The rate limiter will throw a [[yii\web\TooManyRequestsHttpException]] when the rate limit is exceeded.

You may configure the rate limiter as follows in your REST controller classes:

```
public function behaviors()
{
    $behaviors = parent::behaviors();
    $behaviors['rateLimiter']['enableRateLimitHeaders'] = false;
    return $behaviors;
}
```

When rate limiting is enabled, by default every response will be sent with the following HTTP headers containing the current rate limiting information:

- `X-Rate-Limit-Limit`, the maximum number of requests allowed with a time period
- `X-Rate-Limit-Remaining`, the number of remaining requests in the current time period
- `X-Rate-Limit-Reset`, the number of seconds to wait in order to get the maximum number of allowed requests

You may disable these headers by configuring [[yii\filters\RateLimiter::enableRateLimitHeaders]] to be `false`, as shown in the above code example.

Versioning

A good API is *versioned*: changes and new features are implemented in new versions of the API instead of continually altering just one version. Unlike Web applications, with which you have full control of both the client-side and server-side code, APIs are meant to be used by clients beyond your control. For this reason, backward compatibility (BC) of the APIs should be maintained whenever possible. If a change that may break BC is necessary, you should introduce it in new version of the API, and bump up the version number. Existing clients can continue to use the old, working version of the API; and new or upgraded clients can get the new functionality in the new API version.

> Tip: Refer to Semantic Versioning for more information on designing API version numbers.

One common way to implement API versioning is to embed the version number in the API URLs. For example, `http://example.com/v1/users` stands for the `/users` endpoint of API version 1.

Another method of API versioning, which has gained momentum recently, is to put the version number in the HTTP request headers. This is typically done through the `Accept` header:

```
// via a parameter
Accept: application/json; version=v1
// via a vendor content type
Accept: application/vnd.company.myapp-v1+json
```

Both methods have their pros and cons, and there are a lot of debates about each approach. Below you'll see a practical strategy for API versioning that is a mix of these two methods:

- Put each major version of API implementation in a separate module whose ID is the major version number (e.g. `v1`, `v2`). Naturally, the API URLs will contain major version numbers.
- Within each major version (and thus within the corresponding module), use the `Accept` HTTP request header to determine the minor version number and write conditional code to respond to the minor versions accordingly.

For each module serving a major version, the module should include the resource and controller classes serving that specific version. To better separate code responsibility, you may keep a common set of base resource and controller classes, and subclass them in each

individual version module. Within the subclasses, implement the concrete code such as `Model::fields()`.

Your code may be organized like the following:

```
api/
    common/
        controllers/
            UserController.php
            PostController.php
        models/
            User.php
            Post.php
    modules/
        v1/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
            Module.php
        v2/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
            Module.php
```

Your application configuration would look like:

```
return [
    'modules' => [
        'v1' => [
            'class' => 'app\modules\v1\Module',
        ],
        'v2' => [
            'class' => 'app\modules\v2\Module',
        ],
    ],
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'enableStrictParsing' => true,
            'showScriptName' => false,
            'rules' => [
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v1/user', 'v1/post']
],
                ['class' => 'yii\rest\UrlRule', 'controller' => ['v2/user', 'v2/post']
],
            ],
        ],
    ],
];
```

As a result of the above code, `http://example.com/v1/users` will return the list of users in version 1, while `http://example.com/v2/users` will return version 2 users.

Thanks to modules, the code for different major versions can be well isolated. But modules make it still possible to reuse code across the modules via common base classes and other shared resources.

To deal with minor version numbers, you may take advantage of the content negotiation feature provided by the [[yii\filters\ContentNegotiator|contentNegotiator]] behavior. The `contentNegotiator` behavior will set the [[yii\web\Response::acceptParams]] property when it determines which content type to support.

For example, if a request is sent with the HTTP header `Accept: application/json; version=v1`, after content negotiation, [[yii\web\Response::acceptParams]] will contain the value `['version' => 'v1']`.

Based on the version information in `acceptParams`, you may write conditional code in places such as actions, resource classes, serializers, etc. to provide the appropriate functionality.

Since minor versions by definition require maintaining backward compatibility, hopefully there would not be many version checks in your code. Otherwise, chances are that you may need to create a new major version.

Error Handling

When handling a RESTful API request, if there is an error in the user request or if something unexpected happens on the server, you may simply throw an exception to notify the user that something went wrong. If you can identify the cause of the error (e.g., the requested resource does not exist), you should consider throwing an exception along with a proper HTTP status code (e.g., [[yii\web\NotFoundHttpException]] represents a 404 status code). Yii will send the response along with the corresponding HTTP status code and text. Yii will also include the serialized representation of the exception in the response body. For example:

```
HTTP/1.1 404 Not Found
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

The following list summarizes the HTTP status codes that are used by the Yii REST framework:

- `200` : OK. Everything worked as expected.
- `201` : A resource was successfully created in response to a `POST` request. The `Location` header contains the URL pointing to the newly created resource.
- `204` : The request was handled successfully and the response contains no body content (like a `DELETE` request).
- `304` : The resource was not modified. You can use the cached version.
- `400` : Bad request. This could be caused by various actions by the user, such as providing invalid JSON data in the request body, providing invalid action parameters, etc.
- `401` : Authentication failed.
- `403` : The authenticated user is not allowed to access the specified API endpoint.
- `404` : The requested resource does not exist.
- `405` : Method not allowed. Please check the `Allow` header for the allowed HTTP methods.

- `415` : Unsupported media type. The requested content type or version number is invalid.
- `422` : Data validation failed (in response to a `POST` request, for example). Please check the response body for detailed error messages.
- `429` : Too many requests. The request was rejected due to rate limiting.
- `500` : Internal server error. This could be caused by internal program errors.

Customizing Error Response

Sometimes you may want to customize the default error response format. For example, instead of relying on using different HTTP statuses to indicate different errors, you would like to always use 200 as HTTP status and enclose the actual HTTP status code as part of the JSON structure in the response, like shown in the following,

```
HTTP/1.1 200 OK
Date: Sun, 02 Mar 2014 05:31:43 GMT
Server: Apache/2.2.26 (Unix) DAV/2 PHP/5.4.20 mod_ssl/2.2.26 OpenSSL/0.9.8y
Transfer-Encoding: chunked
Content-Type: application/json; charset=UTF-8

{
    "success": false,
    "data": {
        "name": "Not Found Exception",
        "message": "The requested resource was not found.",
        "code": 0,
        "status": 404
    }
}
```

To achieve this goal, you can respond to the `beforeSend` event of the `response` component in the application configuration:

```
return [
    // ...
    'components' => [
        'response' => [
            'class' => 'yii\web\Response',
            'on beforeSend' => function ($event) {
                $response = $event->sender;
                if ($response->data !== null && Yii::$app->request->get('suppress_resp
onse_code')) {
                    $response->data = [
                        'success' => $response->isSuccessful,
                        'data' => $response->data,
                    ];
                    $response->statusCode = 200;
                }
            },
        ],
    ],
];
```

The above code will reformat the response (for both successful and failed responses) as explained when `suppress_response_code` is passed as a `GET` parameter.

Testing

Testing is an important part of software development. Whether we are aware of it or not, we conduct testing continuously. For example, when we write a class in PHP, we may debug it step by step or simply use `echo` or `die` statements to verify the implementation works according to our initial plan. In the case of a web application, we're entering some test data in forms to ensure the page interacts with us as expected.

The testing process could be automated so that each time when we need to verify something, we just need to call up the code that does it for us. The code that verifies the result matches what we've planned is called *test* and the process of its creation and further execution is known as *automated testing*, which is the main topic of these testing chapters.

Developing with tests

Test-Driven Development (TDD) and Behavior-Driven Development (BDD) are approaches of developing software by describing behavior of a piece of code or the whole feature as a set of scenarios or tests before writing actual code and only then creating the implementation that allows these tests to pass verifying that intended behavior is achieved.

The process of developing a feature is the following:

- Create a new test that describes a feature to be implemented.
- Run the new test and make sure it fails. It is expected since there's no implementation yet.
- Write simple code to make the new test pass.
- Run all tests and make sure they all pass.
- Improve code and make sure tests are still OK.

After it's done the process is repeated again for another feature or improvement. If the existing feature is to be changed, tests should be changed as well.

> Tip: If you feel that you are losing time doing a lot of small and simple iterations, try covering more by your test scenario so you do more before executing tests again. If you're debugging too much, try doing the opposite.

The reason to create tests before doing any implementation is that it allows us to focus on what we want to achieve and fully dive into "how to do it" afterwards. Usually it leads to better abstractions and easier test maintenance when it comes to feature adjustments or less coupled components.

So to sum up the pros of such an approach are the following:

- Keeps you focused on one thing at a time which results in improved planning and implementation.
- Results in test-covering more features in greater detail i.e. if tests are OK most likely nothing's broken.

In the long term it usually gives you a good time-saving effect.

> Tip: If you want to know more about the principles for gathering software requirements and modeling the subject matter it's good to learn Domain Driven Development (DDD).

When and how to test

While the test first approach described above makes sense for long term and relatively complex projects it could be overkill for simpler ones. There are some indicators of when it's appropriate:

- Project is already large and complex.
- Project requirements are starting to get complex. Project grows constantly.
- Project is meant to be long term.
- The cost of the failure is too high.

There's nothing wrong in creating tests covering behavior of existing implementation.

- Project is a legacy one to be gradually renewed.
- You've got a project to work on and it has no tests.

In some cases any form of automated testing could be overkill:

- Project is simple and isn't getting anymore complex.
- It's a one-time project that will no longer be worked on.

Still if you have time it's good to automate testing in these cases as well.

Further reading

- Test Driven Development: By Example / Kent Beck. ISBN: 0321146530.

Testing environment setup

> Note: This section is under development.

Yii 2 has officially maintained integration with `Codeception` testing framework that allows you to create the following test types:

- Unit testing - verifies that a single unit of code is working as expected;
- Functional testing - verifies scenarios from a user's perspective via browser emulation;
- Acceptance testing - verifies scenarios from a user's perspective in a browser.

Yii provides ready to use test sets for all three test types in both `yii2-basic` and `yii2-advanced` project templates.

In order to run tests you need to install Codeception. You can install it either locally - for particular project only, or globally - for your development machine.

For the local installation use following commands:

```
composer require "codeception/codeception=2.1.*"
composer require "codeception/specify=*"
composer require "codeception/verify=*"
```

For the global installation you will need to use `global` directive:

```
composer global require "codeception/codeception=2.1.*"
composer global require "codeception/specify=*"
composer global require "codeception/verify=*"
```

If you've never used Composer for global packages before, run `composer global status`. It should output:

```
Changed current directory to <directory>
```

Then add `<directory>/vendor/bin` to you `PATH` environment variable. Now we're able to use `codecept` from command line globally.

> Note: global installation allows you use Codeception for all projects you are working on your development machine and allows running `codecept` shell command globally without specifying path. However, such approach may be inappropriate, for example, if 2 different projects require different versions of Codeception installed. For the simplicity all shell commands related to the tests running around this guide are written assuming Codeception has been installed globally.

Unit Tests

> Note: This section is under development.

A unit test verifies that a single unit of code is working as expected. In object-oriented programming, the most basic code unit is a class. A unit test thus mainly needs to verify that each of the class interface methods works properly. That is, given different input parameters, the test verifies the method returns expected results. Unit tests are usually developed by people who write the classes being tested.

Unit testing in Yii is built on top of PHPUnit and, optionally, Codeception so it's recommended to go through their docs:

- PHPUnit docs starting from chapter 2.
- Codeception Unit Tests.

Running basic and advanced template tests

If you've started with advanced template, please refer to "testing" guide for more details about running tests.

If you've started with basic template, check its README "testing" section.

Framework unit tests

If you want to run unit tests for Yii framework itself follow "Getting started with Yii 2 development".

Functional Tests

> Note: This section is under development.

- Codeception Functional Tests

Running basic and advanced template functional tests

Please refer to instructions provided in `apps/advanced/tests/README.md` and `apps/basic/tests/README.md`.

Acceptance Tests

> Note: This section is under development.

- Codeception Acceptance Tests

Running basic and advanced template acceptance tests

Please refer to instructions provided in `apps/advanced/tests/README.md` and `apps/basic/tests/README.md`.

Fixtures

Fixtures are an important part of testing. Their main purpose is to set up the environment in a fixed/known state so that your tests are repeatable and run in an expected way. Yii provides a fixture framework that allows you to define your fixtures precisely and use them easily.

A key concept in the Yii fixture framework is the so-called *fixture object*. A fixture object represents a particular aspect of a test environment and is an instance of [[yii\test\Fixture]] or its child class. For example, you may use `UserFixture` to make sure the user DB table contains a fixed set of data. You load one or multiple fixture objects before running a test and unload them when finishing.

A fixture may depend on other fixtures, specified via its [[yii\test\Fixture::depends]] property. When a fixture is being loaded, the fixtures it depends on will be automatically loaded BEFORE the fixture; and when the fixture is being unloaded, the dependent fixtures will be unloaded AFTER the fixture.

Defining a Fixture

To define a fixture, create a new class by extending [[yii\test\Fixture]] or [[yii\test\ActiveFixture]]. The former is best suited for general purpose fixtures, while the latter has enhanced features specifically designed to work with database and ActiveRecord.

The following code defines a fixture about the `User` ActiveRecord and the corresponding user table.

```
<?php
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserFixture extends ActiveFixture
{
    public $modelClass = 'app\models\User';
}
```

> Tip: Each `ActiveFixture` is about preparing a DB table for testing purpose. You may specify the table by setting either the [[yii\test\ActiveFixture::tableName]] property or the [[yii\test\ActiveFixture::modelClass]] property. If the latter, the table name will be taken from the `ActiveRecord` class specified by `modelClass`.
>
> Note: [[yii\test\ActiveFixture]] is only suited for SQL databases. For NoSQL databases, Yii provides the following `ActiveFixture` classes:
>
> - Mongo DB: [[yii\mongodb\ActiveFixture]]
> - Elasticsearch: [[yii\elasticsearch\ActiveFixture]] (since version 2.0.2)

The fixture data for an `ActiveFixture` fixture is usually provided in a file located at `FixturePath/data/TableName.php`, where `FixturePath` stands for the directory containing the fixture class file, and `TableName` is the name of the table associated with the fixture. In the example above, the file should be `@app/tests/fixtures/data/user.php`. The data file should return an array of data rows to be inserted into the user table. For example,

```
<?php
return [
    'user1' => [
        'username' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' => '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/iK0r3jRuwQEs2ldRu.a2',
    ],
    'user2' => [
        'username' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' => '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6viYG5xJExU6',
    ],
];
```

You may give an alias to a row so that later in your test, you may refer to the row via the alias. In the above example, the two rows are aliased as `user1` and `user2`, respectively.

Also, you do not need to specify the data for auto-incremental columns. Yii will automatically fill the actual values into the rows when the fixture is being loaded.

> Tip: You may customize the location of the data file by setting the [[yii\test\ActiveFixture::dataFile]] property. You may also override [[yii\test\ActiveFixture::getData()]] to provide the data.

As we described earlier, a fixture may depend on other fixtures. For example, a `UserProfileFixture` may need to depends on `UserFixture` because the user profile table contains a foreign key pointing to the user table. The dependency is specified via the

[[yii\test\Fixture::depends]] property, like the following,

```
namespace app\tests\fixtures;

use yii\test\ActiveFixture;

class UserProfileFixture extends ActiveFixture
{
    public $modelClass = 'app\models\UserProfile';
    public $depends = ['app\tests\fixtures\UserFixture'];
}
```

The dependency also ensures, that the fixtures are loaded and unloaded in a well defined order. In the above example `UserFixture` will always be loaded before `UserProfileFixture` to ensure all foreign key references exist and will be unloaded after `UserProfileFixture` has been unloaded for the same reason.

In the above, we have shown how to define a fixture about a DB table. To define a fixture not related with DB (e.g. a fixture about certain files and directories), you may extend from the more general base class [[yii\test\Fixture]] and override the [[yii\test\Fixture::load()|load()]] and [[yii\test\Fixture::unload()|unload()]] methods.

Using Fixtures

If you are using Codeception to test your code, you should consider using the `yii2-codeception` extension which has built-in support for loading and accessing fixtures. If you are using other testing frameworks, you may use [[yii\test\FixtureTrait]] in your test cases to achieve the same goal.

In the following we will describe how to write a `UserProfile` unit test class using `yii2-codeception`.

In your unit test class extending [[yii\codeception\DbTestCase]] or [[yii\codeception\TestCase]], declare which fixtures you want to use in the [[yii\test\FixtureTrait::fixtures()|fixtures()]] method. For example,

```
namespace app\tests\unit\models;

use yii\codeception\DbTestCase;
use app\tests\fixtures\UserProfileFixture;

class UserProfileTest extends DbTestCase
{
    public function fixtures()
    {
        return [
            'profiles' => UserProfileFixture::className(),
        ];
    }

    // ...test methods...
}
```

The fixtures listed in the `fixtures()` method will be automatically loaded before running every test method in the test case and unloaded after finishing every test method. And as we described before, when a fixture is being loaded, all its dependent fixtures will be automatically loaded first. In the above example, because `UserProfileFixture` depends on `UserFixture`, when running any test method in the test class, two fixtures will be loaded sequentially: `UserFixture` and `UserProfileFixture`.

When specifying fixtures in `fixtures()`, you may use either a class name or a configuration array to refer to a fixture. The configuration array will let you customize the fixture properties when the fixture is loaded.

You may also assign an alias to a fixture. In the above example, the `UserProfileFixture` is aliased as `profiles`. In the test methods, you may then access a fixture object using its alias. For example, `$this->profiles` will return the `UserProfileFixture` object.

Because `UserProfileFixture` extends from `ActiveFixture`, you may further use the following syntax to access the data provided by the fixture:

```
// returns the data row aliased as 'user1'
$row = $this->profiles['user1'];
// returns the UserProfile model corresponding to the data row aliased as 'user1'
$profile = $this->profiles('user1');
// traverse every data row in the fixture
foreach ($this->profiles as $row) ...
```

> Info: `$this->profiles` is still of `UserProfileFixture` type. The above access features are implemented through PHP magic methods.

Defining and Using Global Fixtures

The fixtures described above are mainly used by individual test cases. In most cases, you also need some global fixtures that are applied to ALL or many test cases. An example is [[yii\test\InitDbFixture]] which does two things:

- Perform some common initialization tasks by executing a script located at `@app/tests/fixtures/initdb.php`;
- Disable the database integrity check before loading other DB fixtures, and re-enable it after other DB fixtures are unloaded.

Using global fixtures is similar to using non-global ones. The only difference is that you declare these fixtures in [[yii\codeception\TestCase::globalFixtures()]] instead of `fixtures()`. When a test case loads fixtures, it will first load global fixtures and then non-global ones.

By default, [[yii\codeception\DbTestCase]] already declares `InitDbFixture` in its `globalFixtures()` method. This means you only need to work with `@app/tests/fixtures/initdb.php` if you want to do some initialization work before each test. You may otherwise simply focus on developing each individual test case and the corresponding fixtures.

Organizing Fixture Classes and Data Files

By default, fixture classes look for the corresponding data files under the `data` folder which is a sub-folder of the folder containing the fixture class files. You can follow this convention when working with simple projects. For big projects, chances are that you often need to switch different data files for the same fixture class for different tests. We thus recommend that you organize the data files in a hierarchical way that is similar to your class namespaces. For example,

```
# under folder tests\unit\fixtures

data\
    components\
        fixture_data_file1.php
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
    models\
        fixture_data_file1.php
        fixture_data_file2.php
        ...
        fixture_data_fileN.php
# and so on
```

In this way you will avoid collision of fixture data files between tests and use them as you need.

> Note: In the example above fixture files are named only for example purpose. In real life you should name them according to which fixture class your fixture classes are extending from. For example, if you are extending from [[yii\test\ActiveFixture]] for DB fixtures, you should use DB table names as the fixture data file names; If you are extending from [[yii\mongodb\ActiveFixture]] for MongoDB fixtures, you should use collection names as the file names.

The similar hierarchy can be used to organize fixture class files. Instead of using `data` as the root directory, you may want to use `fixtures` as the root directory to avoid conflict with the data files.

Summary

> Note: This section is under development.

In the above, we have described how to define and use fixtures. Below we summarize the typical workflow of running unit tests related with DB:

1. Use `yii migrate` tool to upgrade your test database to the latest version;
2. Run a test case:
    - Load fixtures: clean up the relevant DB tables and populate them with fixture data;
    - Perform the actual test;
    - Unload fixtures.
3. Repeat Step 2 until all tests finish.

To be cleaned up below

Managing Fixtures

> Note: This section is under development.
>
> todo: this tutorial may be merged with the above part of test-fixtures.md

Fixtures are important part of testing. Their main purpose is to populate you with data that needed by testing different cases. With this data using your tests becoming more efficient and useful.

Yii supports fixtures via the `yii fixture` command line tool. This tool supports:

- Loading fixtures to different storage such as: RDBMS, NoSQL, etc;
- Unloading fixtures in different ways (usually it is clearing storage);
- Auto-generating fixtures and populating it with random data.

Fixtures format

Fixtures are objects with different methods and configurations, refer to official documentation on them. Lets assume we have fixtures data to load:

```
#users.php file under fixtures data path, by default @tests\unit\fixtures\data

return [
    [
        'name' => 'Chase',
        'login' => 'lmayert',
        'email' => 'strosin.vernice@jerde.com',
        'auth_key' => 'K3nF70it7tzNsHddEiq0BZ0i-OU8S3xV',
        'password' => '$2y$13$WSyE5hHsG1rWN2jV8LRHzubilrCLI5Ev/iK0r3jRuwQEs2ldRu.a2',
    ],
    [
        'name' => 'Celestine',
        'login' => 'napoleon69',
        'email' => 'aileen.barton@heaneyschumm.com',
        'auth_key' => 'dZlXsVnIDgIzFgX4EduAqkEPuphhOh9q',
        'password' => '$2y$13$kkgpvJ8lnjKo8RuoR30ay.RjDf15bMcHIF7Vz1zz/6viYG5xJExU6',
    ],
];
```

If we are using fixture that loads data into database then these rows will be applied to `users` table. If we are using nosql fixtures, for example `mongodb` fixture, then this data will be applied to `users` mongodb collection. In order to learn about implementing various

loading strategies and more, refer to official documentation. Above fixture example was auto-generated by `yii2-faker` extension, read more about it in these section. Fixture classes name should not be plural.

Loading fixtures

Fixture classes should be suffixed by `Fixture` class. By default fixtures will be searched under `tests\unit\fixtures` namespace, you can change this behavior with config or command options. You can exclude some fixtures due load or unload by specifying `-` before its name like `-User`.

To load fixture, run the following command:

```
yii fixture/load <fixture_name>
```

The required `fixture_name` parameter specifies a fixture name which data will be loaded. You can load several fixtures at once. Below are correct formats of this command:

```
// load `User` fixture
yii fixture/load User

// same as above, because default action of "fixture" command is "load"
yii fixture User

// load several fixtures
yii fixture "User, UserProfile"

// load all fixtures
yii fixture/load "*"

// same as above
yii fixture "*"

// load all fixtures except ones
yii fixture "*, -DoNotLoadThisOne"

// load fixtures, but search them in different namespace. By default namespace is: tes
ts\unit\fixtures.
yii fixture User --namespace='alias\my\custom\namespace'

// load global fixture `some\name\space\CustomFixture` before other fixtures will be l
oaded.
// By default this option is set to `InitDbFixture` to disable/enable integrity checks
. You can specify several
// global fixtures separated by comma.
yii fixture User --globalFixtures='some\name\space\Custom'
```

Unloading fixtures

To unload fixture, run the following command:

```
// unload Users fixture, by default it will clear fixture storage (for example "users"
 table, or "users" collection if this is mongodb fixture).
yii fixture/unload User

// Unload several fixtures
yii fixture/unload "User, UserProfile"

// unload all fixtures
yii fixture/unload "*"

// unload all fixtures except ones
yii fixture/unload "*, -DoNotUnloadThisOne"
```

Same command options like: `namespace`, `globalFixtures` also can be applied to this command.

Configure Command Globally

While command line options allow us to configure the migration command on-the-fly, sometimes we may want to configure the command once for all. For example you can configure different migration path as follows:

```
'controllerMap' => [
    'fixture' => [
        'class' => 'yii\console\controllers\FixtureController',
        'namespace' => 'myalias\some\custom\namespace',
        'globalFixtures' => [
            'some\name\space\Foo',
            'other\name\space\Bar'
        ],
    ],
]
```

Auto-generating fixtures

Yii also can auto-generate fixtures for you based on some template. You can generate your fixtures with different data on different languages and formats. This feature is done by Faker library and `yii2-faker` extension. See extension guide for more docs.

Creating your own Application structure

> Note: This section is under development.

While the basic and advanced project templates are great for most of your needs, you may want to create your own project template with which to start your projects.

Project templates in Yii are simply repositories containing a `composer.json` file, and registered as a Composer package. Any repository can be identified as a Composer package, making it installable via `create-project` Composer command.

Since it's a bit too much to start building your entire template from scratch, it is better to use one of the built-in templates as a base. Let's use the basic template here.

Clone the Basic Template

The first step is to clone the basic Yii template's Git repository:

```
git clone git@github.com:yiisoft/yii2-app-basic.git
```

Then wait for the repository to be downloaded to your computer. Since the changes made to the template won't be pushed back, you can delete the `.git` directory and all of its contents from the download.

Modify the Files

Next, you'll want to modify the `composer.json` to reflect your template. Change the `name`, `description`, `keywords`, `homepage`, `license`, and `support` values to describe your new template. Also adjust the `require`, `require-dev`, `suggest`, and other options to match your template's requirements.

> Note: In the `composer.json` file, use the `writable` parameter under `extra` to specify per file permissions to be set after an application is created using the template.

Next, actually modify the structure and contents of the application as you would like the default to be. Finally, update the README file to be applicable to your template.

Make a Package

With the template defined, create a Git repository from it, and push your files there. If you're going to open source your template, Github is the best place to host it. If you intend to keep your template non-collaborative, any Git repository site will do.

Next, you need to register your package for Composer's sake. For public templates, the package should be registered at Packagist. For private templates, it is a bit more tricky to register the package. For instructions, see the Composer documentation.

Use the Template

That's all that's required to create a new Yii project template. Now you can create projects using your template:

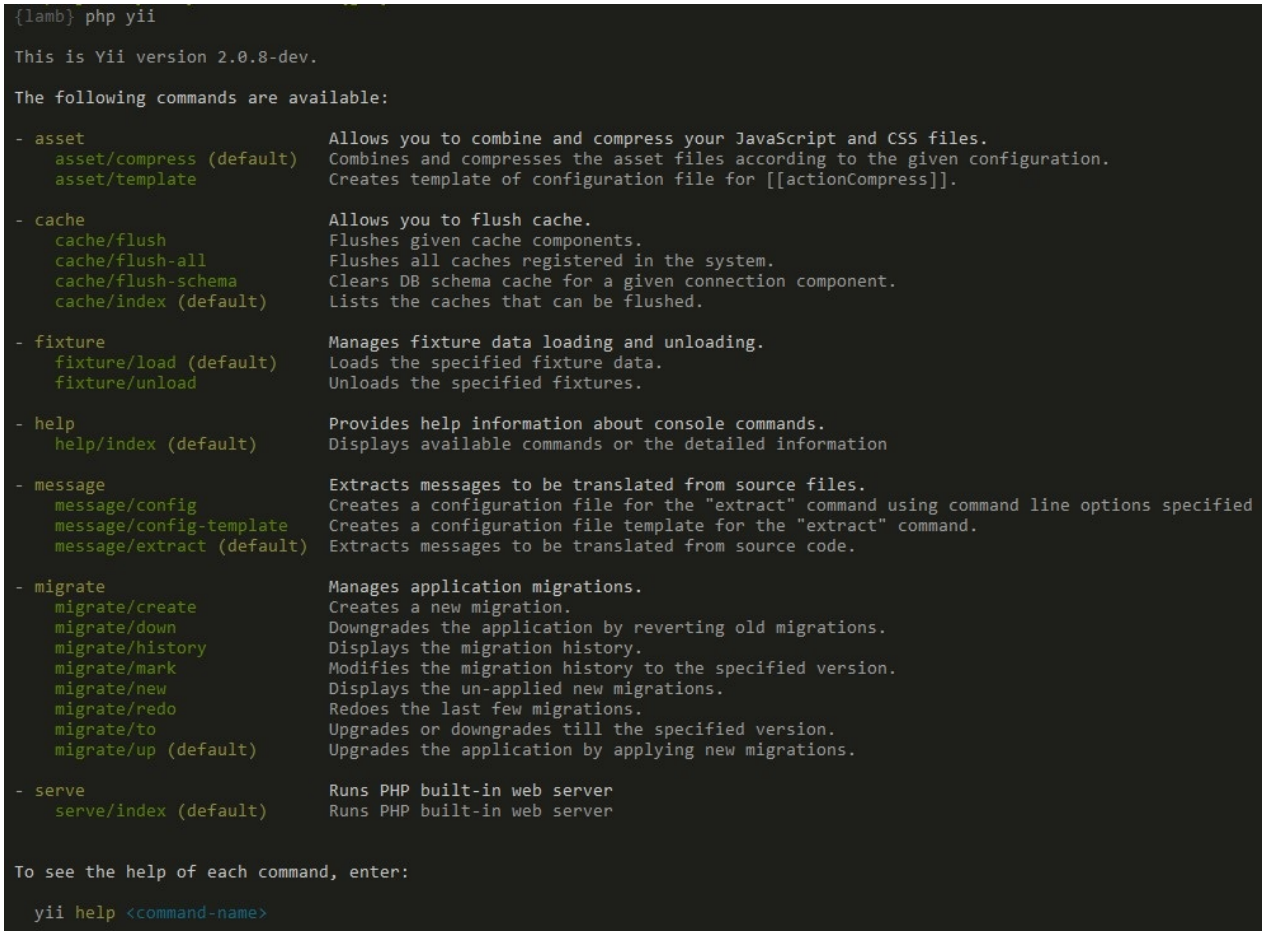
```
composer global require "fxp/composer-asset-plugin:^1.2.0"
composer create-project --prefer-dist --stability=dev mysoft/yii2-app-coolone new-proj
ect
```

Console applications

Besides the rich features for building web applications, Yii also has full-featured support for console applications which are mainly used to create background and maintenance tasks that need to be performed for a website.

The structure of console applications is very similar to a Yii web application. It consists of one or more [[yii\console\Controller]] classes, which are often referred to as *commands* in the console environment. Each controller can also have one or more actions, just like web controllers.

Both project templates already have a console application with them. You can run it by calling the `yii` script, which is located in the base directory of the repository. This will give you a list of available commands when you run it without any further parameters:



As you can see in the screenshot, Yii has already defined a set of commands that are available by default:

- [[yii\console\controllers\AssetController|AssetController]] - Allows you to combine and compress your JavaScript and CSS files. You can learn more about this command in

the Assets Section.
- [[yii\console\controllers\CacheController|CacheController]] - Allows you to flush application caches.
- [[yii\console\controllers\FixtureController|FixtureController]] - Manages fixture data loading and unloading for testing purposes. This command is described in more detail in the Testing Section about Fixtures.
- [[yii\console\controllers\HelpController|HelpController]] - Provides help information about console commands, this is the default command and prints what you have seen in the above output.
- [[yii\console\controllers\MessageController|MessageController]] - Extracts messages to be translated from source files. To learn more about this command, please refer to the I18N Section.
- [[yii\console\controllers\MigrateController|MigrateController]] - Manages application migrations. Database migrations are described in more detail in the Database Migration Section.
- [[yii\console\controllers\ServeController|ServeController]] - Allows you run PHP built-in web server.

Usage

You execute a console controller action using the following syntax:

```
yii <route> [--option1=value1 --option2=value2 ... argument1 argument2 ...]
```

In the above, `<route>` refers to the route to the controller action. The options will populate the class properties and arguments are the parameters of the action method.

For example, the [[yii\console\controllers\MigrateController::actionUp()|MigrateController::actionUp()]] with [[yii\console\controllers\MigrateController::$migrationTable|MigrateController::$migrationTable]] set to `migrations` and a limit of 5 migrations can be called like so:

```
yii migrate/up 5 --migrationTable=migrations
```

> Note: When using `*` in console, don't forget to quote it as `"*"` in order to avoid executing it as a shell glob that will be replaced by all file names of the current directory.

The entry script

The console application entry script is equivalent to the `index.php` bootstrap file used for the web application. The console entry script is typically called `yii`, and located in your application's root directory. It contains code like the following:

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 */

defined('YII_DEBUG') or define('YII_DEBUG', true);
defined('YII_ENV') or define('YII_ENV', 'dev');

require(__DIR__ . '/vendor/autoload.php');
require(__DIR__ . '/vendor/yiisoft/yii2/Yii.php');

$config = require(__DIR__ . '/config/console.php');

$application = new yii\console\Application($config);
$exitCode = $application->run();
exit($exitCode);
```

This script will be created as part of your application; you're free to edit it to suit your needs. The `YII_DEBUG` constant can be set to `false` if you do not want to see a stack trace on error, and/or if you want to improve the overall performance. In both basic and advanced application templates, the console application entry script has debugging enabled by default to provide a more developer-friendly environment.

Configuration

As can be seen in the code above, the console application uses its own configuration file, named `console.php`. In this file you should configure various application components and properties for the console application in particular.

If your web application and console application share a lot of configuration parameters and values, you may consider moving the common parts into a separate file, and including this file in both of the application configurations (web and console). You can see an example of this in the advanced project template.

> Tip: Sometimes, you may want to run a console command using an application configuration that is different from the one specified in the entry script. For example, you may want to use the `yii migrate` command to upgrade your test databases, which are configured in each individual test suite. To change the configuration dynamically, simply specify a custom application configuration file via the `appconfig` option when executing the command:
>
> ```
> yii <route> --appconfig=path/to/config.php ...
> ```

Console command completion

Auto-completion of command arguments is a useful thing when working with the shell. Since version 2.0.11, the `./yii` command provides auto completion for the Bash and ZSH out of the box.

Bash completion

Make sure bash completion is installed. For most of installations it is available by default.

Place the completion script in `/etc/bash_completion.d/` :

```
 curl -L https://raw.githubusercontent.com/yiisoft/yii2/master/contrib/completion/bash
/yii -o /etc/bash_completion.d/yii
```

For temporary usage you can put the file into the current directory and include it in the current session via `source yii` . If globally installed you may need to restart the terminal or `source ~/.bashrc` to activate it.

Check the Bash Manual for other ways of including completion script to your environment.

ZSH completion

Put the completion script in directory for completions, using e.g. `~/.zsh/completion/`

```
mkdir -p ~/.zsh/completion
curl -L https://raw.githubusercontent.com/yiisoft/yii2/master/contrib/completion/zsh/_
yii -o ~/.zsh/completion/_yii
```

Include the directory in the `$fpath` , e.g. by adding it to `~/.zshrc`

```
fpath=(~/.zsh/completion $fpath)
```

Make sure `compinit` is loaded or do it by adding in `~/.zshrc`

```
autoload -Uz compinit && compinit -i
```

Then reload your shell

```
exec $SHELL -l
```

Creating your own console commands

Console Controller and Action

A console command is defined as a controller class extending from [[yii\console\Controller]]. In the controller class, you define one or more actions that correspond to sub-commands of the controller. Within each action, you write code that implements the appropriate tasks for that particular sub-command.

When running a command, you need to specify the route to the controller action. For example, the route `migrate/create` invokes the sub-command that corresponds to the [[yii\console\controllers\MigrateController::actionCreate()|MigrateController::actionCreate()]] action method. If a route offered during execution does not contain an action ID, the default action will be executed (as with a web controller).

Options

By overriding the [[yii\console\Controller::options()]] method, you can specify options that are available to a console command (controller/actionID). The method should return a list of the controller class's public properties. When running a command, you may specify the value of an option using the syntax `--optionName=optionValue`. This will assign `optionValue` to the `optionName` property of the controller class.

If the default value of an option is of an array type and you set this option while running the command, the option value will be converted into an array by splitting the input string on any commas.

Options Aliases

Since version 2.0.8 console command provides [[yii\console\Controller::optionAliases()]] method to add aliases for options.

To define an alias, override [[yii\console\Controller::optionAliases()]] in your controller, for example:

```
namespace app\commands;

use yii\console\Controller;

class HelloController extends Controller
{
    public $message;

    public function options($actionID)
    {
        return ['message'];
    }

    public function optionAliases()
    {
        return ['m' => 'message'];
    }

    public function actionIndex()
    {
        echo $this->message . "\n";
    }
}
```

Now, you can use the following syntax to run the command:

```
yii hello -m=hello
```

Arguments

Besides options, a command can also receive arguments. The arguments will be passed as the parameters to the action method corresponding to the requested sub-command. The first argument corresponds to the first parameter, the second corresponds to the second, and so on. If not enough arguments are provided when the command is called, the corresponding parameters will take the declared default values, if defined. If no default value is set, and no value is provided at runtime, the command will exit with an error.

You may use the `array` type hint to indicate that an argument should be treated as an array. The array will be generated by splitting the input string on commas.

The following example shows how to declare arguments:

```
class ExampleController extends \yii\console\Controller
{
    // The command "yii example/create test" will call "actionCreate('test')"
    public function actionCreate($name) { ... }

    // The command "yii example/index city" will call "actionIndex('city', 'name')"
    // The command "yii example/index city id" will call "actionIndex('city', 'id')"
    public function actionIndex($category, $order = 'name') { ... }

    // The command "yii example/add test" will call "actionAdd(['test'])"
    // The command "yii example/add test1,test2" will call "actionAdd(['test1', 'test2
'])"
    public function actionAdd(array $name) { ... }
}
```

Exit Code

Using exit codes is a best practice for console application development. Conventionally, a command returns `0` to indicate that everything is OK. If the command returns a number greater than zero, that's considered to be indicative of an error. The number returned will be the error code, potentially usable to find out details about the error. For example `1` could stand generally for an unknown error and all codes above would be reserved for specific cases: input errors, missing files, and so forth.

To have your console command return an exit code, simply return an integer in the controller action method:

```
public function actionIndex()
{
    if (/* some problem */) {
        echo "A problem occurred!\n";
        return 1;
    }
    // do something
    return 0;
}
```

There are some predefined constants you can use:

- [[yii\console\Controller::EXIT_CODE_NORMAL|Controller::EXIT_CODE_NORMAL]] with value of `0`;
- [[yii\console\Controller::EXIT_CODE_ERROR|Controller::EXIT_CODE_ERROR]] with value of `1`.

It's a good practice to define meaningful constants for your controller in case you have more error code types.

Formatting and colors

Yii console supports formatted output that is automatically degraded to non-formatted one if it's not supported by terminal running the command.

Outputting formatted strings is simple. Here's how to output some bold text:

```
$this->stdout("Hello?\n", Console::BOLD);
```

If you need to build string dynamically combining multiple styles it's better to use [[yii\helpers\Console::ansiFormat()|ansiFormat()]]:

```
$name = $this->ansiFormat('Alex', Console::FG_YELLOW);
echo "Hello, my name is $name.";
```

Core Validators

Yii provides a set of commonly used core validators, found primarily under the `yii\validators` namespace. Instead of using lengthy validator class names, you may use *aliases* to specify the use of these core validators. For example, you can use the alias `required` to refer to the [[yii\validators\RequiredValidator]] class:

```
public function rules()
{
    return [
        [['email', 'password'], 'required'],
    ];
}
```

The [[yii\validators\Validator::builtInValidators]] property declares all supported validator aliases.

In the following, we will describe the main usage and properties of every core validator.

[[yii\validators\BooleanValidator|boolean]]

```
[
    // checks if "selected" is either 0 or 1, regardless of data type
    ['selected', 'boolean'],

    // checks if "deleted" is of boolean type, either true or false
    ['deleted', 'boolean', 'trueValue' => true, 'falseValue' => false, 'strict' => true
],
]
```

This validator checks if the input value is a boolean.

- `trueValue` : the value representing `true` . Defaults to `'1'` .
- `falseValue` : the value representing `false` . Defaults to `'0'` .
- `strict` : whether the type of the input value should match that of `trueValue` and `falseValue` . Defaults to `false` .

> Note: Because data input submitted via HTML forms are all strings, you normally should leave the [[yii\validators\BooleanValidator::strict|strict]] property as `false` .

[[yii\captcha\CaptchaValidator|captcha]]

```
[
    ['verificationCode', 'captcha'],
]
```

This validator is usually used together with [[yii\captcha\CaptchaAction]] and [[yii\captcha\Captcha]] to make sure an input is the same as the verification code displayed by [[yii\captcha\Captcha|CAPTCHA]] widget.

- `caseSensitive`: whether the comparison of the verification code is case sensitive. Defaults to `false`.
- `captchaAction`: the route corresponding to the [[yii\captcha\CaptchaAction|CAPTCHA action]] that renders the CAPTCHA image. Defaults to `'site/captcha'`.
- `skipOnEmpty`: whether the validation can be skipped if the input is empty. Defaults to `false`, which means the input is required.

[[yii\validators\CompareValidator|compare]]

```
[
    // validates if the value of "password" attribute equals to that of "password_repe
at"
    ['password', 'compare'],

    // same as above but with explicitly specifying the attribute to compare with
    ['password', 'compare', 'compareAttribute' => 'password_repeat'],

    // validates if age is greater than or equal to 30
    ['age', 'compare', 'compareValue' => 30, 'operator' => '>=', 'type' => 'number'],
]
```

This validator compares the specified input value with another one and make sure if their relationship is as specified by the `operator` property.

- `compareAttribute`: the name of the attribute whose value should be compared with. When the validator is being used to validate an attribute, the default value of this property would be the name of the attribute suffixed with `_repeat`. For example, if the attribute being validated is `password`, then this property will default to `password_repeat`.
- `compareValue`: a constant value that the input value should be compared with. When both of this property and `compareAttribute` are specified, this property will take precedence.
- `operator`: the comparison operator. Defaults to `==`, meaning checking if the input

value is equal to that of `compareAttribute` or `compareValue`. The following operators are supported:
    - `==` : check if two values are equal. The comparison is done is non-strict mode.
    - `===` : check if two values are equal. The comparison is done is strict mode.
    - `!=` : check if two values are NOT equal. The comparison is done is non-strict mode.
    - `!==` : check if two values are NOT equal. The comparison is done is strict mode.
    - `>` : check if value being validated is greater than the value being compared with.
    - `>=` : check if value being validated is greater than or equal to the value being compared with.
    - `<` : check if value being validated is less than the value being compared with.
    - `<=` : check if value being validated is less than or equal to the value being compared with.
- `type` : The default comparison type is '[[yii\validators\CompareValidator::TYPE_STRING|string]]', which means the values are compared byte by byte. When comparing numbers, make sure to set the [[yii\validators\CompareValidator::$type|$type]] to '[[yii\validators\CompareValidator::TYPE_NUMBER|number]]' to enable numeric comparison.

Comparing date values

The compare validator can only be used to compare strings and numbers. If you need to compare values like dates you have two options. For comparing a date against a fixed value, you can simply use the [[yii\validators\DateValidator|date]] validator and specify its [[yii\validators\DateValidator::$min|$min]] or [[yii\validators\DateValidator::$max|$max]] property. If you need to compare two dates entered in the form, e.g. a `fromDate` and a `toDate` field, you can use a combination of compare and date validator like the following:

```
['fromDate', 'date', 'timestampAttribute' => 'fromDate'],
['toDate', 'date', 'timestampAttribute' => 'toDate'],
['fromDate', 'compare', 'compareAttribute' => 'toDate', 'operator' => '<', 'enableClie
ntValidation' => false],
```

As validators are executed in the order they are specified this will first validate that the values entered in `fromDate` and `toDate` are valid date values and if so, they will be converted into a machine readable format. Afterwards these two values are compared with the compare validator. Client validation is not enabled as this will only work on the server-side because the date validator currently does not provide client validation, so [[yii\validators\CompareValidator::$enableClientValidation|$enableClientValidation]] is set to `false` on the compare validator too.

[[yii\validators\DateValidator|date]]

The [[yii\validators\DateValidator|date]] validator comes with three different shortcuts:

```
[
    [['from_date', 'to_date'], 'date'],
    [['from_datetime', 'to_datetime'], 'datetime'],
    [['some_time'], 'time'],
]
```

This validator checks if the input value is a date, time or datetime in a proper format. Optionally, it can convert the input value into a UNIX timestamp or other machine readable format and store it in an attribute specified via [[yii\validators\DateValidator::timestampAttribute|timestampAttribute]].

- `format`: the date/time format that the value being validated should be in. This can be a date time pattern as described in the ICU manual. Alternatively this can be a string prefixed with `php:` representing a format that can be recognized by the PHP `Datetime` class. Please refer to http://php.net/manual/en/datetime.createfromformat.php on supported formats. If this is not set, it will take the value of `Yii::$app->formatter->dateFormat`. See the [[yii\validators\DateValidator::$format|API documentation]] for more details.
- `timestampAttribute`: the name of the attribute to which this validator may assign the UNIX timestamp converted from the input date/time. This can be the same attribute as the one being validated. If this is the case, the original value will be overwritten with the timestamp value after validation. See "Handling date input with the DatePicker" for a usage example.

  Since version 2.0.4, a format and timezone can be specified for this attribute using [[yii\validators\DateValidator::$timestampAttributeFormat|$timestampAttributeFormat]] and [[yii\validators\DateValidator::$timestampAttributeTimeZone|$timestampAttributeTimeZone]].

  Note, that when using `timestampAttribute`, the input value will be converted to a unix timestamp, which by definition is in UTC, so a conversion from the [[yii\validators\DateValidator::timeZone|input time zone]] to UTC will be performed.
- Since version 2.0.4 it is also possible to specify a [[yii\validators\DateValidator::$min|minimum]] or [[yii\validators\DateValidator::$max|maximum]] timestamp.

In case the input is optional you may also want to add a default value filter in addition to the date validator to ensure empty input is stored as `null`. Otherwise you may end up with dates like `0000-00-00` in your database or `1970-01-01` in the input field of a date picker.

```
[
    [['from_date', 'to_date'], 'default', 'value' => null],
    [['from_date', 'to_date'], 'date'],
],
```

[[yii\validators\DefaultValueValidator|default]]

```
[
    // set "age" to be null if it is empty
    ['age', 'default', 'value' => null],

    // set "country" to be "USA" if it is empty
    ['country', 'default', 'value' => 'USA'],

    // assign "from" and "to" with a date 3 days and 6 days from today, if they are em
pty
    [['from', 'to'], 'default', 'value' => function ($model, $attribute) {
        return date('Y-m-d', strtotime($attribute === 'to' ? '+3 days' : '+6 days'));
    }],
]
```

This validator does not validate data. Instead, it assigns a default value to the attributes being validated if the attributes are empty.

- `value`: the default value or a PHP callable that returns the default value which will be assigned to the attributes being validated if they are empty. The signature of the PHP callable should be as follows,

```
function foo($model, $attribute) {
    // ... compute $value ...
    return $value;
}
```

> Info: How to determine if a value is empty or not is a separate topic covered in the Empty Values section.

[[yii\validators\NumberValidator|double]]

```
[
    // checks if "salary" is a double number
    ['salary', 'double'],
]
```

This validator checks if the input value is a double number. It is equivalent to the number validator.

- `max` : the upper limit (inclusive) of the value. If not set, it means the validator does not check the upper limit.
- `min` : the lower limit (inclusive) of the value. If not set, it means the validator does not check the lower limit.

[[yii\validators\EachValidator|each]]

> Info: This validator has been available since version 2.0.4.

```
[
    // checks if every category ID is an integer
    ['categoryIDs', 'each', 'rule' => ['integer']],
]
```

This validator only works with an array attribute. It validates if *every* element of the array can be successfully validated by a specified validation rule. In the above example, the `categoryIDs` attribute must take an array value and each array element will be validated by the `integer` validation rule.

- `rule` : an array specifying a validation rule. The first element in the array specifies the class name or the alias of the validator. The rest of the name-value pairs in the array are used to configure the validator object.
- `allowMessageFromRule` : whether to use the error message returned by the embedded validation rule. Defaults to `true` . If `false` , it will use `message` as the error message.

> Note: If the attribute value is not an array, it is considered validation fails and the `message` will be returned as the error message.

[[yii\validators\EmailValidator|email]]

```
[
    // checks if "email" is a valid email address
    ['email', 'email'],
]
```

This validator checks if the input value is a valid email address.

- `allowName`: whether to allow name in the email address (e.g. `John Smith <john.smith@example.com>`). Defaults to `false`.
- `checkDNS`, whether to check whether the email's domain exists and has either an A or MX record. Be aware that this check may fail due to temporary DNS problems, even if the email address is actually valid. Defaults to `false`.
- `enableIDN`, whether the validation process should take into account IDN (internationalized domain names). Defaults to `false`. Note that in order to use IDN validation you have to install and enable the `intl` PHP extension, or an exception would be thrown.

[[yii\validators\ExistValidator|exist]]

```
[
    // a1 needs to exist in the column represented by the "a1" attribute
    ['a1', 'exist'],

    // a1 needs to exist, but its value will use a2 to check for the existence
    ['a1', 'exist', 'targetAttribute' => 'a2'],

    // a1 and a2 need to exist together, and they both will receive error message
    [['a1', 'a2'], 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a1 and a2 need to exist together, only a1 will receive error message
    ['a1', 'exist', 'targetAttribute' => ['a1', 'a2']],

    // a1 needs to exist by checking the existence of both a2 and a3 (using a1 value)
    ['a1', 'exist', 'targetAttribute' => ['a2', 'a1' => 'a3']],

    // a1 needs to exist. If a1 is an array, then every element of it must exist.
    ['a1', 'exist', 'allowArray' => true],
]
```

This validator checks if the input value can be found in a table column represented by an Active Record attribute. You can use `targetAttribute` to specify the Active Record attribute and `targetClass` the corresponding Active Record class. If you do not specify them, they will take the values of the attribute and the model class being validated.

You can use this validator to validate against a single column or multiple columns (i.e., the combination of multiple attribute values should exist).

- `targetClass`: the name of the Active Record class that should be used to look for the input value being validated. If not set, the class of the model currently being validated will be used.
- `targetAttribute`: the name of the attribute in `targetClass` that should be used to validate the existence of the input value. If not set, it will use the name of the attribute currently being validated. You may use an array to validate the existence of multiple columns at the same time. The array values are the attributes that will be used to validate the existence, while the array keys are the attributes whose values are to be validated. If the key and the value are the same, you can just specify the value.
- `filter`: additional filter to be applied to the DB query used to check the existence of the input value. This can be a string or an array representing the additional query condition (refer to [[yii\db\Query::where()]] on the format of query condition), or an anonymous function with the signature `function ($query)`, where `$query` is the [[yii\db\Query|Query]] object that you can modify in the function.
- `allowArray`: whether to allow the input value to be an array. Defaults to `false`. If this property is `true` and the input is an array, then every element of the array must exist in the target column. Note that this property cannot be set `true` if you are validating against multiple columns by setting `targetAttribute` as an array.

[[yii\validators\FileValidator|file]]

```
[
    // checks if "primaryImage" is an uploaded image file in PNG, JPG or GIF format.
    // the file size must be less than 1MB
    ['primaryImage', 'file', 'extensions' => ['png', 'jpg', 'gif'], 'maxSize' => 1024*
1024],
]
```

This validator checks if the input is a valid uploaded file.

- `extensions`: a list of file name extensions that are allowed to be uploaded. This can be either an array or a string consisting of file extension names separated by space or comma (e.g. "gif, jpg"). Extension names are case-insensitive. Defaults to `null`, meaning all file name extensions are allowed.
- `mimeTypes`: a list of file MIME types that are allowed to be uploaded. This can be either an array or a string consisting of file MIME types separated by space or comma (e.g. "image/jpeg, image/png"). The wildcard mask with the special character `*` can be used to match groups of mime types. For example `image/*` will pass all mime types, that

begin with `image/` (e.g. `image/jpeg`, `image/png`). Mime type names are case-insensitive. Defaults to `null`, meaning all MIME types are allowed. For more details, please refer to common media types.

- `minSize`: the minimum number of bytes required for the uploaded file. Defaults to `null`, meaning no lower limit.
- `maxSize`: the maximum number of bytes allowed for the uploaded file. Defaults to `null`, meaning no upper limit.
- `maxFiles`: the maximum number of files that the given attribute can hold. Defaults to 1, meaning the input must be a single uploaded file. If it is greater than 1, then the input must be an array consisting of at most `maxFiles` number of uploaded files.
- `checkExtensionByMimeType`: whether to check the file extension by the file's MIME type. If the extension produced by MIME type check differs from the uploaded file extension, the file will be considered as invalid. Defaults to `true`, meaning perform such check.

`FileValidator` is used together with [[yii\web\UploadedFile]]. Please refer to the Uploading Files section for complete coverage about uploading files and performing validation about the uploaded files.

[[yii\validators\FilterValidator|filter]]

```
[
    // trim "username" and "email" inputs
    [['username', 'email'], 'filter', 'filter' => 'trim', 'skipOnArray' => true],

    // normalize "phone" input
    ['phone', 'filter', 'filter' => function ($value) {
        // normalize phone input here
        return $value;
    }],
]
```

This validator does not validate data. Instead, it applies a filter on the input value and assigns it back to the attribute being validated.

- `filter`: a PHP callback that defines a filter. This can be a global function name, an anonymous function, etc. The function signature must be `function ($value) { return $newValue; }`. This property must be set.
- `skipOnArray`: whether to skip the filter if the input value is an array. Defaults to `false`. Note that if the filter cannot handle array input, you should set this property to be `true`. Otherwise some PHP error might occur.

> Tip: If you want to trim input values, you may directly use the trim validator.
>
> Tip: There are many PHP functions that have the signature expected for the `filter` callback. For example to apply type casting (using e.g. intval, boolval, ...) to ensure a specific type for an attribute, you can simply specify the function names of the filter without the need to wrap them in a closure:
>
> ```
> ['property', 'filter', 'filter' => 'boolval'],
> ['property', 'filter', 'filter' => 'intval'],
> ```

[[yii\validators\ImageValidator|image]]

```
[
    // checks if "primaryImage" is a valid image with proper size
    ['primaryImage', 'image', 'extensions' => 'png, jpg',
        'minWidth' => 100, 'maxWidth' => 1000,
        'minHeight' => 100, 'maxHeight' => 1000,
    ],
]
```

This validator checks if the input value represents a valid image file. It extends from the file validator and thus inherits all its properties. Besides, it supports the following additional properties specific for image validation purpose:

- `minWidth` : the minimum width of the image. Defaults to `null` , meaning no lower limit.
- `maxWidth` : the maximum width of the image. Defaults to `null` , meaning no upper limit.
- `minHeight` : the minimum height of the image. Defaults to `null` , meaning no lower limit.
- `maxHeight` : the maximum height of the image. Defaults to `null` , meaning no upper limit.

[[yii\validators\IpValidator|ip]]

```
[
    // checks if "ip_address" is a valid IPv4 or IPv6 address
    ['ip_address', 'ip'],

    // checks if "ip_address" is a valid IPv6 address or subnet,
    // value will be expanded to full IPv6 notation.
    ['ip_address', 'ip', 'ipv4' => false, 'subnet' => null, 'expandIPv6' => true],

    // checks if "ip_address" is a valid IPv4 or IPv6 address,
    // allows negation character `!` at the beginning
    ['ip_address', 'ip', 'negation' => true],
]
```

The validator checks if the attribute value is a valid IPv4/IPv6 address or subnet. It also may change attribute's value if normalization or IPv6 expansion is enabled.

The validator has such configuration options:

- `ipv4` : whether the validating value can be an IPv4 address. Defaults to `true` .
- `ipv6` : whether the validating value can be an IPv6 address. Defaults to `true` .
- `subnet` : whether the address can be an IP with CIDR subnet, like `192.168.10.0/24`
  - `true` - the subnet is required, addresses without CIDR will be rejected
  - `false` - the address can not have the CIDR
  - `null` - the CIDR is optional

  Defaults to `false` .
- `normalize` : whether to add the CIDR prefix with the smallest length (32 for IPv4 and 128 for IPv6) to an address without it. Works only when `subnet` is not `false` . For example:
  - `10.0.1.5` will normalized to `10.0.1.5/32`
  - `2008:db0::1` will be normalized to `2008:db0::1/128`

  Defaults to `false` .
- `negation` : whether the validation address can have a negation character `!` at the beginning. Defaults to `false` .
- `expandIPv6` : whether to expand an IPv6 address to the full notation format. For example, `2008:db0::1` will be expanded to `2008:0db0:0000:0000:0000:0000:0000:0001` . Defaults to `false` .
- `ranges` : array of IPv4 or IPv6 ranges that are allowed or forbidden.

  When the array is empty, or the option is not set, all the IP addresses are allowed. Otherwise, the rules are checked sequentially until the first match is found. IP address is forbidden, when it has not matched any of the rules.

For example:

```
[
     'client_ip', 'ip', 'ranges' => [
         '192.168.10.128'
         '!192.168.10.0/24',
         'any' // allows any other IP addresses
     ]
]
```

In this example, access is allowed for all the IPv4 and IPv6 addresses excluding `192.168.10.0/24` subnet. IPv4 address `192.168.10.128` is also allowed, because it is listed before the restriction.

- `networks`: array of network aliases, that can be used in `ranges`. Format of array:
  - key - alias name
  - value - array of strings. String can be a range, IP address or another alias. String can be negated with `!` (independent of `negation` option).

    The following aliases are defined by default:

  - `*`: `any`
  - `any`: `0.0.0.0/0, ::/0`
  - `private`: `10.0.0.0/8, 172.16.0.0/12, 192.168.0.0/16, fd00::/8`
  - `multicast`: `224.0.0.0/4, ff00::/8`
  - `linklocal`: `169.254.0.0/16, fe80::/10`
  - `localhost`: `127.0.0.0/8', ::1`
  - `documentation`: `192.0.2.0/24, 198.51.100.0/24, 203.0.113.0/24, 2001:db8::/32`
  - `system`: `multicast, linklocal, localhost, documentation`

> Info: This validator has been available since version 2.0.7.

[[yii\validators\RangeValidator|in]]

```
[
    // checks if "level" is 1, 2 or 3
    ['level', 'in', 'range' => [1, 2, 3]],
]
```

This validator checks if the input value can be found among the given list of values.

- `range`: a list of given values within which the input value should be looked for.

- `strict` : whether the comparison between the input value and the given values should be strict (both the type and value must be the same). Defaults to `false` .
- `not` : whether the validation result should be inverted. Defaults to `false` . When this property is set `true` , the validator checks if the input value is NOT among the given list of values.
- `allowArray` : whether to allow the input value to be an array. When this is `true` and the input value is an array, every element in the array must be found in the given list of values, or the validation would fail.

[[yii\validators\NumberValidator|integer]]

```
[
    // checks if "age" is an integer
    ['age', 'integer'],
]
```

This validator checks if the input value is an integer.

- `max` : the upper limit (inclusive) of the value. If not set, it means the validator does not check the upper limit.
- `min` : the lower limit (inclusive) of the value. If not set, it means the validator does not check the lower limit.

[[yii\validators\RegularExpressionValidator|match]]

```
[
    // checks if "username" starts with a letter and contains only word characters
    ['username', 'match', 'pattern' => '/^[a-z]\w*$/i']
]
```

This validator checks if the input value matches the specified regular expression.

- `pattern` : the regular expression that the input value should match. This property must be set, or an exception will be thrown.
- `not` : whether to invert the validation result. Defaults to `false` , meaning the validation succeeds only if the input value matches the pattern. If this is set `true` , the validation is considered successful only if the input value does NOT match the pattern.

[[yii\validators\NumberValidator|number]]

```
[
    // checks if "salary" is a number
    ['salary', 'number'],
]
```

This validator checks if the input value is a number. It is equivalent to the double validator.

- `max` : the upper limit (inclusive) of the value. If not set, it means the validator does not check the upper limit.
- `min` : the lower limit (inclusive) of the value. If not set, it means the validator does not check the lower limit.

[[yii\validators\RequiredValidator|required]]

```
[
    // checks if both "username" and "password" are not empty
    [['username', 'password'], 'required'],
]
```

This validator checks if the input value is provided and not empty.

- `requiredValue` : the desired value that the input should be. If not set, it means the input should not be empty.
- `strict` : whether to check data types when validating a value. Defaults to `false` . When `requiredValue` is not set, if this property is `true` , the validator will check if the input value is not strictly `null` ; If this property is `false` , the validator will use a loose rule to determine a value is empty or not. When `requiredValue` is set, the comparison between the input and `requiredValue` will also check data types if this property is `true` .

> Info: How to determine if a value is empty or not is a separate topic covered in the Empty Values section.

[[yii\validators\SafeValidator|safe]]

```
[
    // marks "description" to be a safe attribute
    ['description', 'safe'],
]
```

This validator does not perform data validation. Instead, it is used to mark an attribute to be a safe attribute.

[[yii\validators\StringValidator|string]]

```
[
    // checks if "username" is a string whose length is between 4 and 24
    ['username', 'string', 'length' => [4, 24]],
]
```

This validator checks if the input value is a valid string with certain length.

- `length` : specifies the length limit of the input string being validated. This can be specified in one of the following forms:
  - an integer: the exact length that the string should be of;
  - an array of one element: the minimum length of the input string (e.g. `[8]`). This will overwrite `min` .
  - an array of two elements: the minimum and maximum lengths of the input string (e.g. `[8, 128]`). This will overwrite both `min` and `max` .
- `min` : the minimum length of the input string. If not set, it means no minimum length limit.
- `max` : the maximum length of the input string. If not set, it means no maximum length limit.
- `encoding` : the encoding of the input string to be validated. If not set, it will use the application's [[yii\base\Application::charset|charset]] value which defaults to `UTF-8` .

[[yii\validators\FilterValidator|trim]]

```
[
    // trims the white spaces surrounding "username" and "email"
    [['username', 'email'], 'trim'],
]
```

This validator does not perform data validation. Instead, it will trim the surrounding white spaces around the input value. Note that if the input value is an array, it will be ignored by this validator.

[[yii\validators\UniqueValidator|unique]]

```
[
    // a1 needs to be unique in the column represented by the "a1" attribute
    ['a1', 'unique'],

    // a1 needs to be unique, but column a2 will be used to check the uniqueness of th
e a1 value
    ['a1', 'unique', 'targetAttribute' => 'a2'],

    // a1 and a2 need to be unique together, and they both will receive error message
    [['a1', 'a2'], 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 and a2 need to be unique together, only a1 will receive error message
    ['a1', 'unique', 'targetAttribute' => ['a1', 'a2']],

    // a1 needs to be unique by checking the uniqueness of both a2 and a3 (using a1 va
lue)
    ['a1', 'unique', 'targetAttribute' => ['a2', 'a1' => 'a3']],
]
```

This validator checks if the input value is unique in a table column. It only works with Active Record model attributes. It supports validation against either a single column or multiple columns.

- `targetClass` : the name of the Active Record class that should be used to look for the input value being validated. If not set, the class of the model currently being validated will be used.
- `targetAttribute` : the name of the attribute in `targetClass` that should be used to validate the uniqueness of the input value. If not set, it will use the name of the attribute currently being validated. You may use an array to validate the uniqueness of multiple columns at the same time. The array values are the attributes that will be used to validate the uniqueness, while the array keys are the attributes whose values are to be validated. If the key and the value are the same, you can just specify the value.
- `filter` : additional filter to be applied to the DB query used to check the uniqueness of the input value. This can be a string or an array representing the additional query condition (refer to [[yii\db\Query::where()]] on the format of query condition), or an anonymous function with the signature `function ($query)` , where `$query` is the [[yii\db\Query|Query]] object that you can modify in the function.

[[yii\validators\UrlValidator|url]]

```
[
    // checks if "website" is a valid URL. Prepend "http://" to the "website" attribute

    // if it does not have a URI scheme
    ['website', 'url', 'defaultScheme' => 'http'],
]
```

This validator checks if the input value is a valid URL.

- `validSchemes` : an array specifying the URI schemes that should be considered valid. Defaults to `['http', 'https']` , meaning both `http` and `https` URLs are considered to be valid.
- `defaultScheme` : the default URI scheme to be prepended to the input if it does not have the scheme part. Defaults to `null` , meaning do not modify the input value.
- `enableIDN` : whether the validator should take into account IDN (internationalized domain names). Defaults to `false` . Note that in order to use IDN validation you have to install and enable the `intl` PHP extension, otherwise an exception would be thrown.

Internationalization

Internationalization (I18N) refers to the process of designing a software application so that it can be adapted to various languages and regions without engineering changes. For Web applications, this is of particular importance because the potential users may be worldwide. Yii offers a full spectrum of I18N features that support message translation, view translation, date and number formatting.

Locale and Language

Locale

Locale is a set of parameters that defines the user's language, country and any special variant preferences that the user wants to see in their user interface. It is usually identified by an ID consisting of a language ID and a region ID.

For example, the ID `en-US` stands for the locale of "English and the United States".

For consistency reasons, all locale IDs used in Yii applications should be canonicalized to the format of `ll-cc`, where `ll` is a two- or three-letter lowercase language code according to ISO-639 and `cc` is a two-letter country code according to ISO-3166. More details about locale can be found in the documentation of the ICU project.

Language

In Yii, we often use the term "language" to refer to a locale.

A Yii application uses two kinds of languages:

- [[yii\base\Application::$sourceLanguage|source language]]: This refers to the language in which the text messages in the source code are written.
- [[yii\base\Application::$language|target language]]: This is the language that should be used to display content to end users.

The so-called message translation service mainly translates a text message from source language to target language.

Configuration

You can configure application languages in the "application configuration" like the following:

```
return [
    // set target language to be Russian
    'language' => 'ru-RU',

    // set source language to be English
    'sourceLanguage' => 'en-US',

    ......
];
```

The default value for the [[yii\base\Application::$sourceLanguage|source language]] is `en-US`, meaning US English. It is **recommended** that you keep this default value unchanged. Usually it is much easier to find people who can translate from "English to other languages" than from "non-English to non-English".

You often need to set the [[yii\base\Application::$language|target language]] dynamically based on different factors, such as the language preference of end users. Instead of configuring it in the application configuration, you can use the following statement to change the target language:

```
// change target language to Chinese
\Yii::$app->language = 'zh-CN';
```

> Tip: If your source language varies among different parts of your code, you can override the source language for different message sources, which are described in the next section.

Message Translation

From source language to target language

The message translation service translates a text message from one language (usually the [[yii\base\Application::$sourceLanguage|source language]]) to another (usually the [[yii\base\Application::$language|target language]]).

It does the translation by looking up the message to be translated in a message source which stores the original messages and the translated messages. If the message is found, the corresponding translated message will be returned; otherwise the original message will be returned untranslated.

How to implement

To use the message translation service, you mainly need to do the following work:

1. Wrap every text message that needs to be translated in a call to the [[Yii::t()]] method.
2. Configure one or multiple message sources in which the message translation service can look for translated messages.
3. Let the translators translate messages and store them in the message source(s).

1. Wrap a text message

The method [[Yii::t()]] can be used like the following,

```
echo \Yii::t('app', 'This is a string to translate!');
```

where the second parameter refers to the text message to be translated, while the first parameter refers to the name of the category which is used to categorize the message.

2. Configure one or multiple message sources

The [[Yii::t()]] method will call the `i18n` application component `translate` method to perform the actual translation work. The component can be configured in the application configuration as follows,

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                //'basePath' => '@app/messages',
                //'sourceLanguage' => 'en-US',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
            ],
        ],
    ],
],
```

In the above code, a message source supported by [[yii\i18n\PhpMessageSource]] is being configured.

Category wildcards with `*` symbol

The pattern `app*` indicates that all message categories whose names start with `app` should be translated using this message source.

3. Let the translators translate messages and store them in the message source(s)

The [[yii\i18n\PhpMessageSource]] class uses PHP files with a simple PHP array to store message translations. These files contain a map of the messages in `source language` to the translation in the `target language`.

> Info: You can automatically generate these PHP files by using the `message` command, which will be introduced later in this chapter.

Each PHP file corresponds to the messages of a single category. By default, the file name should be the same as the category name. Example for `app/messages/nl-NL/main.php:`

```
<?php

/**
* Translation map for nl-NL
*/
return [
    'welcome' => 'welkom'
];
```

File mapping

You may configure [[yii\i18n\PhpMessageSource::fileMap|fileMap]] to map a category to a PHP file with a different naming approach.

In the above example, the category `app/error` is mapped to the PHP file `@app/messages/ru-RU/error.php` (assuming `ru-RU` is the target language). However, without this configuration the category would be mapped to `@app/messages/ru-RU/app/error.php` instead.

Other storage types

Besides storing the messages in PHP files, you may also use the following message sources to store translated messages in different storage:

- [[yii\i18n\GettextMessageSource]] uses GNU Gettext MO or PO files to maintain translated messages.
- [[yii\i18n\DbMessageSource]] uses a database table to store translated messages.

Message Formatting

When translating a message, you can embed some placeholders and have them replaced by dynamic parameter values. You can even use special placeholder syntax to have the parameter values formatted according to the target language. In this subsection, we will describe different ways of formatting messages.

Message Parameters

In a message to be translated, you can embed one or multiple parameters (also called placeholders) so that they can be replaced by the given values. By giving different sets of values, you can variate the translated message dynamically. In the following example, the placeholder `{username}` in the message `'Hello, {username}!'` will be replaced by `'Alexander'` and `'Qiang'`, respectively.

```
$username = 'Alexander';
// display a translated message with username being "Alexander"
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);

$username = 'Qiang';
// display a translated message with username being "Qiang"
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);
```

While translating a message containing placeholders, you should leave the placeholders as is. This is because the placeholders will be replaced with the actual values when you call `Yii::t()` to translate a message.

You can use either *named placeholders* or *positional placeholders*, but not both, in a single message.

The previous example shows how you can use named placeholders. That is, each placeholder is written in the format of `{name}`, and you provide an associative array whose keys are the placeholder names (without the curly brackets) and whose values are the corresponding values placeholder to be replaced with.

Positional placeholders use zero-based integer sequence as names which are replaced by the provided values according to their positions in the call of `Yii::t()`. In the following example, the positional placeholders `{0}`, `{1}` and `{2}` will be replaced by the values of `$price`, `$count` and `$subtotal`, respectively.

```
$price = 100;
$count = 2;
$subtotal = 200;
echo \Yii::t('app', 'Price: {0}, Count: {1}, Subtotal: {2}', [$price, $count, $subtota
l]);
```

In case of a single positional parameter its value could be specified without wrapping it into array:

```
echo \Yii::t('app', 'Price: {0}', $price);
```

> Tip: In most cases you should use named placeholders. This is because the names will make the translators understand better the whole messages being translated.

Parameter Formatting

You can specify additional formatting rules in the placeholders of a message so that the parameter values can be formatted properly before they replace the placeholders. In the following example, the price parameter value will be treated as a number and formatted as a currency value:

```
$price = 100;
echo \Yii::t('app', 'Price: {0,number,currency}', $price);
```

> Note: Parameter formatting requires the installation of the intl PHP extension.

You can use either the short form or the full form to specify a placeholder with formatting:

```
short form: {name,type}
full form: {name,type,style}
```

> Note: If you need to use special characters such as `{`, `}`, `'`, `#`, wrap them in `'`:
>
> ```
> echo Yii::t('app', "Example of string with ''-escaped characters'': '{' '}' '{tes
> t}' {count,plural,other{''count'' value is # '#{}'}}", ['count' => 3]);
> ```

Complete format is described in the ICU documentation. In the following we will show some common usages.

Number

The parameter value is treated as a number. For example,

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0,number}', $sum);
```

You can specify an optional parameter style as `integer`, `currency`, or `percent`:

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0,number,currency}', $sum);
```

You can also specify a custom pattern to format the number. For example,

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0,number,,000,000000}', $sum);
```

Characters used in the custom format could be found in ICU API reference under "Special Pattern Characters" section.

The value is always formatted according to the locale you are translating to i.e. you cannot change decimal or thousands separators, currency symbol etc. without changing translation locale. If you need to customize these you can use [[yii\i18n\Formatter::asDecimal()]] and [[yii\i18n\Formatter::asCurrency()]].

Date

The parameter value should be formatted as a date. For example,

```
echo \Yii::t('app', 'Today is {0,date}', time());
```

You can specify an optional parameter style as `short`, `medium`, `long`, or `full`:

```
echo \Yii::t('app', 'Today is {0,date,short}', time());
```

You can also specify a custom pattern to format the date value:

```
echo \Yii::t('app', 'Today is {0,date,yyyy-MM-dd}', time());
```

Formatting reference.

Time

The parameter value should be formatted as a time. For example,

```
echo \Yii::t('app', 'It is {0,time}', time());
```

You can specify an optional parameter style as `short`, `medium`, `long`, or `full`:

```
echo \Yii::t('app', 'It is {0,time,short}', time());
```

You can also specify a custom pattern to format the time value:

```
echo \Yii::t('app', 'It is {0,date,HH:mm}', time());
```

Formatting reference.

Spellout

The parameter value should be treated as a number and formatted as a spellout. For example,

```
// may produce "42 is spelled as forty-two"
echo \Yii::t('app', '{n,number} is spelled as {n,spellout}', ['n' => 42]);
```

By default the number is spelled out as cardinal. It could be changed:

```
// may produce "I am forty-seventh agent"
echo \Yii::t('app', 'I am {n,spellout,%spellout-ordinal} agent', ['n' => 47]);
```

Note that there should be no space after `spellout,` and before `%`.

To get a list of options available for locale you're using check "Numbering schemas, Spellout" at http://intl.rmcreative.ru/.

Ordinal

The parameter value should be treated as a number and formatted as an ordinal name. For example,

```
// may produce "You are the 42nd visitor here!"
echo \Yii::t('app', 'You are the {n,ordinal} visitor here!', ['n' => 42]);
```

Ordinal supports more ways of formatting for languages such as Spanish:

```
// may produce 471ª
echo \Yii::t('app', '{n,ordinal,%digits-ordinal-feminine}', ['n' => 471]);
```

Note that there should be no space after `ordinal,` and before `%`.

To get a list of options available for locale you're using check "Numbering schemas, Ordinal" at http://intl.rmcreative.ru/.

Duration

The parameter value should be treated as the number of seconds and formatted as a time duration string. For example,

```
// may produce "You are here for 47 sec. already!"
echo \Yii::t('app', 'You are here for {n,duration} already!', ['n' => 47]);
```

Duration supports more ways of formatting:

```
// may produce 130:53:47
echo \Yii::t('app', '{n,duration,%in-numerals}', ['n' => 471227]);
```

Note that there should be no space after `duration,` and before `%`.

To get a list of options available for locale you're using check "Numbering schemas, Duration" at http://intl.rmcreative.ru/.

Plural

Different languages have different ways to inflect plurals. Yii provides a convenient way for translating messages in different plural forms that works well even for very complex rules. Instead of dealing with the inflection rules directly, it is sufficient to provide the translation of inflected words in certain situations only. For example,

```
// When $n = 0, it may produce "There are no cats!"
// When $n = 1, it may produce "There is one cat!"
// When $n = 42, it may produce "There are 42 cats!"
echo \Yii::t('app', 'There {n,plural,=0{are no cats} =1{is one cat} other{are # cats}}
!', ['n' => $n]);
```

In the plural rule arguments above, `=` means explicit value. So `=0` means exactly zero, `=1` means exactly one. `other` stands for any other value. `#` is replaced with the value of `n` formatted according to target language.

Plural forms can be very complicated in some languages. In the following Russian example, `=1` matches exactly `n = 1` while `one` matches `21` or `101`:

```
Здесь {n,plural,=0{котов нет} =1{есть один кот} one{# кот} few{# кота} many{# котов} o
ther{# кота}}!
```

These `other`, `few`, `many` and other special argument names vary depending on language. To learn which ones you should specify for a particular locale, please refer to "Plural Rules, Cardinal" at http://intl.rmcreative.ru/. Alternatively you can refer to rules reference at unicode.org.

> Note: The above example Russian message is mainly used as a translated message, not an original message, unless you set the [[yii\base\Application::$sourceLanguage|source language]] of your application as `ru-RU` and translating from Russian.
>
> When a translation is not found for an original message specified in `Yii::t()` call, the plural rules for the [[yii\base\Application::$sourceLanguage|source language]] will be applied to the original message.

There's an `offset` parameter for the cases when the string is like the following:

```
$likeCount = 2;
echo Yii::t('app', 'You {likeCount,plural,
    offset: 1
    =0{did not like this}
    =1{liked this}
    one{and one other person liked this}
    other{and # others liked this}
}', [
    'likeCount' => $likeCount
]);

// You and one other person liked this
```

Ordinal selection

The parameter type of `selectordinal` is meant to choose a string based on language rules for ordinals for the locale you are translating to:

```
$n = 3;
echo Yii::t('app', 'You are the {n,selectordinal,one{#st} two{#nd} few{#rd} other{#th}
} visitor', ['n' => $n]);
// For English it outputs:
// You are the 3rd visitor

// Translation
'You are the {n,selectordinal,one{#st} two{#nd} few{#rd} other{#th}} visitor' => 'Вы {
n,selectordinal,other{#-й}} посетитель',

// For Russian translation it outputs:
// Вы 3-й посетитель
```

The format is very close to what's used for plurals. To learn which arguments you should specify for a particular locale, please refer to "Plural Rules, Ordinal" at http://intl.rmcreative.ru/. Alternatively you can refer to rules reference at unicode.org.

Selection

You can use the `select` parameter type to choose a phrase based on the parameter value. For example,

```
// It may produce "Snoopy is a dog and it loves Yii!"
echo \Yii::t('app', '{name} is a {gender} and {gender,select,female{she} male{he} othe
r{it}} loves Yii!', [
    'name' => 'Snoopy',
    'gender' => 'dog',
]);
```

In the expression above, both `female` and `male` are possible parameter values, while `other` handles values that do not match either one of them. Following each possible parameter value, you should specify a phrase and enclose it in a pair of curly brackets.

Specifying default message source

You can specify default message source that will be used as a fallback for category that doesn't match any configured category. You can do that by configuring a wildcard category `*`. In order to do that, add the following to the application config:

```
//configure i18n component

'i18n' => [
    'translations' => [
        '*' => [
            'class' => 'yii\i18n\PhpMessageSource'
        ],
    ],
],
```

Now you can use categories without configuring each one, which is similar to Yii 1.1 behavior. Messages for the category will be loaded from a file under the default translation `basePath` that is `@app/messages` :

```
echo Yii::t('not_specified_category', 'message from unspecified category');
```

The message will be loaded from `@app/messages/<LanguageCode>/not_specified_category.php` .

Translating module messages

If you want to translate the messages for a module and avoid using a single translation file for all the messages, you can do it like the following:

```php
<?php

namespace app\modules\users;

use Yii;

class Module extends \yii\base\Module
{
    public $controllerNamespace = 'app\modules\users\controllers';

    public function init()
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        Yii::$app->i18n->translations['modules/users/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/modules/users/messages',
            'fileMap' => [
                'modules/users/validation' => 'validation.php',
                'modules/users/form' => 'form.php',
                ...
            ],
        ];
    }

    public static function t($category, $message, $params = [], $language = null)
    {
        return Yii::t('modules/users/' . $category, $message, $params, $language);
    }

}
```

In the example above we are using wildcard for matching and then filtering each category per needed file. Instead of using `fileMap`, you can simply use the convention of the category mapping to the same named file. Now you can use `Module::t('validation', 'your custom validation message')` or `Module::t('form', 'some form label')` directly.

Translating widgets messages

The same rule as applied for Modules above can be applied for widgets too, for example:

```php
<?php

namespace app\widgets\menu;

use yii\base\Widget;
use Yii;

class Menu extends Widget
{

    public function init()
    {
        parent::init();
        $this->registerTranslations();
    }

    public function registerTranslations()
    {
        $i18n = Yii::$app->i18n;
        $i18n->translations['widgets/menu/*'] = [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/widgets/menu/messages',
            'fileMap' => [
                'widgets/menu/messages' => 'messages.php',
            ],
        ];
    }

    public function run()
    {
        echo $this->render('index');
    }

    public static function t($category, $message, $params = [], $language = null)
    {
        return Yii::t('widgets/menu/' . $category, $message, $params, $language);
    }

}
```

Instead of using `fileMap` you can simply use the convention of the category mapping to the same named file. Now you can use `Menu::t('messages', 'new messages {messages}', ['{messages}' => 10])` directly.

> Note: For widgets you also can use i18n views, with the same rules as for controllers being applied to them too.

Translating framework messages

Yii comes with the default translation messages for validation errors and some other strings. These messages are all in the category `yii`. Sometimes you want to correct the default framework message translation for your application. In order to do so, configure the `i18n` application component like the following:

```
'i18n' => [
    'translations' => [
        'yii' => [
            'class' => 'yii\i18n\PhpMessageSource',
            'sourceLanguage' => 'en-US',
            'basePath' => '@app/messages'
        ],
    ],
],
```

Now you can place your adjusted translations to `@app/messages/<language>/yii.php`.

Handling missing translations

Even if the translation is missing from the source, Yii will display the requested message content. Such behavior is very convenient in case your raw message is a valid verbose text. However, sometimes it is not enough. You may need to perform some custom processing of the situation, when the requested translation is missing from the source. This can be achieved using the [[yii\i18n\MessageSource::EVENT_MISSING_TRANSLATION|missingTranslation]]-event of [[yii\i18n\MessageSource]].

For example, you may want to mark all the missing translations with something notable, so that they can be easily found at the page. First you need to setup an event handler. This can be done in the application configuration:

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
                'on missingTranslation' => ['app\components\TranslationEventHandler',
'handleMissingTranslation']
            ],
        ],
    ],
],
```

Now you need to implement your own event handler:

```
<?php

namespace app\components;

use yii\i18n\MissingTranslationEvent;

class TranslationEventHandler
{
    public static function handleMissingTranslation(MissingTranslationEvent $event)
    {
        $event->translatedMessage = "@MISSING: {$event->category}.{$event->message} FO
R LANGUAGE {$event->language} @";
    }
}
```

If [[yii\i18n\MissingTranslationEvent::translatedMessage]] is set by the event handler it will be displayed as the translation result.

> Note: each message source handles its missing translations separately. If you are using several message sources and wish them to treat the missing translations in the same way, you should assign the corresponding event handler to each of them.

Using the `message` command

Translations can be stored in [[yii\i18n\PhpMessageSource|php files]], [[yii\i18n\GettextMessageSource|.po files]] or in a [[yii\i18n\DbMessageSource|database]]. See specific classes for additional options.

First of all you need to create a configuration file. Decide where you want to store it and then issue the command

```
./yii message/config-template path/to/config.php
```

Open the created file and adjust the parameters to fit your needs. Pay special attention to:

- `languages` : an array representing what languages your app should be translated to;
- `messagePath` : path where to store message files, which should match the `i18n` 's `basePath` parameter stated in config.

You may also use './yii message/config' command to dynamically generate configuration file with specified options via cli. For example, you can set `languages` and `messagePath` parameters like the following:

```
./yii message/config --languages=de,ja --messagePath=messages path/to/config.php
```

To get list of available options execute next command:

```
./yii help message/config
```

Once you're done with the configuration file you can finally extract your messages with the command:

```
./yii message path/to/config.php
```

Also, you may use options to dynamically change parameters for extraction.

You will then find your files (if you've chosen file based translations) in your `messagePath` directory.

View Translation

Instead of translating individual text messages, sometimes you may want to translate a whole view script. To achieve this goal, simply translate the view and save it under a subdirectory whose name is the same as target language. For example, if you want to translate the view script `views/site/index.php` and the target language is `ru-RU` , you may translate the view and save it as the file `views/site/ru-RU/index.php` . Now whenever you

call [[yii\base\View::renderFile()]] or any method that invoke this method (e.g. [[yii\base\Controller::render()]]) to render the view `views/site/index.php`, it will end up rendering the translated view `views/site/ru-RU/index.php`, instead.

> Note: If the [[yii\base\Application::$language|target language]] is the same as [[yii\base\Application::$sourceLanguage|source language]] original view will be rendered regardless of presence of translated view.

Formatting Date and Number Values

See the Data Formatting section for details.

Setting Up PHP Environment

Yii uses the PHP intl extension to provide most of its I18N features, such as the date and number formatting of the [[yii\i18n\Formatter]] class and the message formatting using [[yii\i18n\MessageFormatter]]. Both classes provide a fallback mechanism when the `intl` extension is not installed. However, the fallback implementation only works well for English target language. So it is highly recommended that you install `intl` when I18N is needed.

The PHP intl extension is based on the ICU library which provides the knowledge and formatting rules for all different locales. Different versions of ICU may produce different formatting result of date and number values. To ensure your website produces the same results across all environments, it is recommended that you install the same version of the `intl` extension (and thus the same version of ICU) in all environments.

To find out which version of ICU is used by PHP, you can run the following script, which will give you the PHP and ICU version being used.

```
<?php
echo "PHP: " . PHP_VERSION . "\n";
echo "ICU: " . INTL_ICU_VERSION . "\n";
echo "ICU Data: " . INTL_ICU_DATA_VERSION . "\n";
```

It is also recommended that you use an ICU version equal or greater than version 49. This will ensure you can use all the features described in this document. For example, an ICU version below 49 does not support using `#` placeholders in plural rules. Please refer to http://site.icu-project.org/download for a complete list of available ICU versions. Note that the version numbering has changed after the 4.8 release (e.g., ICU 4.8, ICU 49, ICU 50, etc.)

Additionally the information in the time zone database shipped with the ICU library may be outdated. Please refer to the ICU manual for details on updating the time zone database. While for output formatting the ICU timezone database is used, the time zone database used by PHP may be relevant too. You can update it by installing the latest version of the pecl package `timezonedb`.

Mailing

> Note: This section is under development.

Yii supports composition and sending of the email messages. However, the framework core provides only the content composition functionality and basic interface. Actual mail sending mechanism should be provided by the extension, because different projects may require its different implementation and it usually depends on the external services and libraries.

For the most common cases you can use yii2-swiftmailer official extension.

Configuration

Mail component configuration depends on the extension you have chosen. In general your application configuration should look like:

```
return [
    //....
    'components' => [
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
    ],
];
```

Basic usage

Once the `mailer` component is configured, you can use the following code to send an email message:

```
Yii::$app->mailer->compose()
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Message subject')
    ->setTextBody('Plain text content')
    ->setHtmlBody('<b>HTML content</b>')
    ->send();
```

In the above example the method `compose()` creates an instance of the mail message, which then is populated and sent. You may put more complex logic in this process if needed:

```
$message = Yii::$app->mailer->compose();
if (Yii::$app->user->isGuest) {
    $message->setFrom('from@domain.com');
} else {
    $message->setFrom(Yii::$app->user->identity->email);
}
$message->setTo(Yii::$app->params['adminEmail'])
    ->setSubject('Message subject')
    ->setTextBody('Plain text content')
    ->send();
```

> Note: each `mailer` extension comes in 2 major classes: `Mailer` and `Message`. `Mailer` always knows the class name and specific of the `Message`. Do not attempt to instantiate `Message` object directly — always use `compose()` method for it.

You may also send several messages at once:

```
$messages = [];
foreach ($users as $user) {
    $messages[] = Yii::$app->mailer->compose()
        // ...
        ->setTo($user->email);
}
Yii::$app->mailer->sendMultiple($messages);
```

Some particular mail extensions may benefit from this approach, using single network message etc.

Composing mail content

Yii allows composition of the actual mail messages content via special view files. By default these files should be located at `@app/mail` path.

Example mail view file content:

```
<?php
use yii\helpers\Html;
use yii\helpers\Url;

/* @var $this \yii\web\View view component instance */
/* @var $message \yii\mail\BaseMessage instance of newly created mail message */

?>
<h2>This message allows you to visit our site home page by one click</h2>
<?= Html::a('Go to home page', Url::home('http')) ?>
```

In order to compose message content via view file simply pass view name to the `compose()` method:

```
Yii::$app->mailer->compose('home-link') // a view rendering result becomes the message
 body here
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Message subject')
    ->send();
```

You may pass additional view parameters to `compose()` method, which will be available inside the view files:

```
Yii::$app->mailer->compose('greetings', [
    'user' => Yii::$app->user->identity,
    'advertisement' => $adContent,
]);
```

You can specify different view files for HTML and plain text message contents:

```
Yii::$app->mailer->compose([
    'html' => 'contact-html',
    'text' => 'contact-text',
]);
```

If you specify view name as a scalar string, its rendering result will be used as HTML body, while plain text body will be composed by removing all HTML entities from HTML one.

View rendering result can be wrapped into the layout, which can be setup using [[yii\mail\BaseMailer::htmlLayout]] and [[yii\mail\BaseMailer::textLayout]]. It will work the same way like layouts in regular web application. Layout can be used to setup mail CSS styles or other shared content:

```
<?php
use yii\helpers\Html;

/* @var $this \yii\web\View view component instance */
/* @var $message \yii\mail\MessageInterface the message being composed */
/* @var $content string main view render result */
?>
<?php $this->beginPage() ?>
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Strict//EN" "http://www.w3.org/TR/xhtml1/
DTD/xhtml1-strict.dtd">
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
    <meta http-equiv="Content-Type" content="text/html; charset=<?= Yii::$app->charset
 ?>" />
    <style type="text/css">
        .heading {...}
        .list {...}
        .footer {...}
    </style>
    <?php $this->head() ?>
</head>
<body>
    <?php $this->beginBody() ?>
    <?= $content ?>
    <div class="footer">With kind regards, <?= Yii::$app->name ?> team</div>
    <?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

File attachment

You can add attachments to message using methods `attach()` and `attachContent()`:

```
$message = Yii::$app->mailer->compose();

// attach file from local file system
$message->attach('/path/to/source/file.pdf');

// create attachment on-the-fly
$message->attachContent('Attachment content', ['fileName' => 'attach.txt', 'contentTyp
e' => 'text/plain']);
```

Embedding images

You can embed images into the message content using `embed()` method. This method returns the attachment id, which should be then used at `img` tag. This method is easy to use when composing message content via view file:

```
Yii::$app->mailer->compose('embed-email', ['imageFileName' => '/path/to/image.jpg'])
    // ...
    ->send();
```

Then inside the view file you can use the following code:

```
<img src="<?= $message->embed($imageFileName); ?>">
```

Testing and debugging

A developer often has to check, what actual emails are sent by the application, what was their content and so on. Such ability is granted by Yii via `yii\mail\BaseMailer::useFileTransport`. If enabled, this option enforces saving mail message data into the local files instead of regular sending. These files will be saved under `yii\mail\BaseMailer::fileTransportPath`, which is `@runtime/mail` by default.

> Note: you can either save the messages to the files or send them to the actual recipients, but can not do both simultaneously.

A mail message file can be opened by a regular text file editor, so you can browse the actual message headers, content and so on. This mechanism may prove itself, while debugging application or running unit test.

> Note: the mail message file content is composed via `\yii\mail\MessageInterface::toString()`, so it depends on the actual mail extension you are using in your application.

Creating your own mail solution

In order to create your own custom mail solution, you need to create 2 classes: one for the `Mailer` and another one for the `Message`. You can use `yii\mail\BaseMailer` and `yii\mail\BaseMessage` as the base classes for your solution. These classes already contain the basic logic, which is described in this guide. However, their usage is not mandatory, it is enough to implement `yii\mail\MailerInterface` and `yii\mail\MessageInterface` interfaces. Then you need to implement all the abstract methods to build your solution.

Performance Tuning

There are many factors affecting the performance of your Web application. Some are environmental, some are related with your code, while some others are related with Yii itself. In this section, we will enumerate most of these factors and explain how you can improve your application performance by adjusting these factors.

Optimizing your PHP Environment

A well configured PHP environment is very important. In order to get maximum performance,

- Use the latest stable PHP version. Major releases of PHP may bring significant performance improvements.
- Enable bytecode caching with Opcache (PHP 5.5 or later) or APC (PHP 5.4). Bytecode caching avoids the time spent in parsing and including PHP scripts for every incoming request.
- Tune `realpath()` cache.

Disabling Debug Mode

When running an application in production, you should disable debug mode. Yii uses the value of a constant named `YII_DEBUG` to indicate whether debug mode should be enabled. When debug mode is enabled, Yii will take extra time to generate and record debugging information.

You may place the following line of code at the beginning of the entry script to disable debug mode:

```
defined('YII_DEBUG') or define('YII_DEBUG', false);
```

> Info: The default value of `YII_DEBUG` is `false`. So if you are certain that you do not change its default value somewhere else in your application code, you may simply remove the above line to disable debug mode.

Using Caching Techniques

You can use various caching techniques to significantly improve the performance of your application. For example, if your application allows users to enter text in Markdown format, you may consider caching the parsed Markdown content to avoid parsing the same Markdown text repeatedly in every request. Please refer to the Caching section to learn about the caching support provided by Yii.

Enabling Schema Caching

Schema caching is a special caching feature that should be enabled whenever you are using Active Record. As you know, Active Record is intelligent enough to detect schema information (e.g. column names, column types, constraints) about a DB table without requiring you to manually describe them. Active Record obtains this information by executing extra SQL queries. By enabling schema caching, the retrieved schema information will be saved in the cache and reused in future requests.

To enable schema caching, configure a `cache` application component to store the schema information and set [[yii\db\Connection::enableSchemaCache]] to be `true` in the application configuration:

```
return [
    // ...
    'components' => [
        // ...
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=mydatabase',
            'username' => 'root',
            'password' => '',
            'enableSchemaCache' => true,

            // Duration of schema cache.
            'schemaCacheDuration' => 3600,

            // Name of the cache component used to store schema information
            'schemaCache' => 'cache',
        ],
    ],
];
```

Combining and Minimizing Assets

A complex Web page often includes many CSS and/or JavaScript asset files. To reduce the number of HTTP requests and the overall download size of these assets, you should consider combining them into one single file and compressing it. This may greatly improve the page loading time and reduce the server load. For more details, please refer to the Assets section.

Optimizing Session Storage

By default session data are stored in files. The implementation is locking a file from opening a session to the point it's closed either by `session_write_close()` (in Yii it could be done as `Yii::$app->session->close()`) or at the end of request. While session file is locked all other requests which are trying to use the same session are blocked i.e. waiting for the initial request to release session file. This is fine for development and probably small projects. But when it comes to handling massive concurrent requests, it is better to use more sophisticated storage, such as database. Yii supports a variety of session storage out of box. You can use these storage by configuring the `session` component in the application configuration like the following,

```
return [
    // ...
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',

            // Set the following if you want to use DB component other than
            // default 'db'.
            // 'db' => 'mydb',

            // To override default session table, set the following
            // 'sessionTable' => 'my_session',
        ],
    ],
];
```

The above configuration uses a database table to store session data. By default, it will use the `db` application component as the database connection and store the session data in the `session` table. You do have to create the `session` table as follows in advance, though,

```
CREATE TABLE session (
    id CHAR(40) NOT NULL PRIMARY KEY,
    expire INTEGER,
    data BLOB
)
```

You may also store session data in a cache by using [[yii\web\CacheSession]]. In theory, you can use any supported cache storage. Note, however, that some cache storage may flush cached data when the storage limit is reached. For this reason, you should mainly use those cache storage that do not enforce storage limit.

If you have Redis on your server, it is highly recommended you use it as session storage by using [[yii\redis\Session]].

Optimizing Databases

Executing DB queries and fetching data from databases are often the main performance bottleneck in a Web application. Although using data caching techniques may alleviate the performance hit, it does not fully solve the problem. When the database contains enormous amounts of data and the cached data is invalid, fetching the latest data could be prohibitively expensive without proper database and query design.

A general technique to improve the performance of DB queries is to create indices for table columns that need to be filtered by. For example, if you need to look for a user record by `username`, you should create an index on `username`. Note that while indexing can make SELECT queries much faster, it will slow down INSERT, UPDATE and DELETE queries.

For complex DB queries, it is recommended that you create database views to save the query parsing and preparation time.

Last but not least, use `LIMIT` in your `SELECT` queries. This avoids fetching an overwhelming amount of data from the database and exhausting the memory allocated to PHP.

Using Plain Arrays

Although Active Record is very convenient to use, it is not as efficient as using plain arrays when you need to retrieve a large amount of data from database. In this case, you may consider calling `asArray()` while using Active Record to query data so that the retrieved data is represented as arrays instead of bulky Active Record objects. For example,

```
class PostController extends Controller
{
    public function actionIndex()
    {
        $posts = Post::find()->limit(100)->asArray()->all();

        return $this->render('index', ['posts' => $posts]);
    }
}
```

In the above code, `$posts` will be populated as an array of table rows. Each row is a plain array. To access the `title` column of the i-th row, you may use the expression `$posts[$i]['title']`.

You may also use DAO to build queries and retrieve data in plain arrays.

Optimizing Composer Autoloader

Because Composer autoloader is used to include most third-party class files, you should consider optimizing it by executing the following command:

```
composer dumpautoload -o
```

Processing Data Offline

When a request involves some resource intensive operations, you should think of ways to perform those operations in offline mode without having users wait for them to finish.

There are two methods to process data offline: pull and push.

In the pull method, whenever a request involves some complex operation, you create a task and save it in a persistent storage, such as database. You then use a separate process (such as a cron job) to pull the tasks and process them. This method is easy to implement, but it has some drawbacks. For example, the task process needs to periodically pull from the task storage. If the pull frequency is too low, the tasks may be processed with great delay, but if the frequency is too high, it will introduce high overhead.

In the push method, you would use a message queue (e.g. RabbitMQ, ActiveMQ, Amazon SQS, etc.) to manage the tasks. Whenever a new task is put on the queue, it will initiate or notify the task handling process to trigger the task processing.

Performance Profiling

You should profile your code to find out the performance bottlenecks and take appropriate measures accordingly. The following profiling tools may be useful:

- Yii debug toolbar and debugger
- Blackfire
- XHProf
- XDebug profiler

Prepare application for scaling

When nothing helps you may try making your application scalabe. A good introduction is provided in Configuring a Yii 2 Application for an Autoscaling Stack. For further reading you may refer to Web apps performance and scaling.

Shared Hosting Environment

Shared hosting environments are often quite limited about configuration and directory structure. Still in most cases you can run Yii 2.0 on a shared hosting environment with a few adjustments.

Deploying a basic project template

Since in a shared hosting environment there's typically only one webroot, use the basic project template if you can. Refer to the Installing Yii chapter and install the basic project template locally. After you have the application working locally, we'll make some adjustments so it can be hosted on your shared hosting server.

Renaming webroot

Connect to your shared host using FTP or by other means. You will probably see something like the following.

```
config
logs
www
```

In the above, `www` is your webserver webroot directory. It could be named differently. Common names are: `www`, `htdocs`, and `public_html`.

The webroot in our basic project template is named `web`. Before uploading the application to your webserver rename your local webroot to match your server, i.e., from `web` to `www`, `public_html` or whatever the name of your hosting webroot.

FTP root directory is writeable

If you can write to the root level directory i.e. where `config`, `logs` and `www` are, then upload `assets`, `commands` etc. as is to the root level directory.

Add extras for webserver

If your webserver is Apache you'll need to add an `.htaccess` file with the following content to `web` (or `public_html` or whatever) (where the `index.php` file is located):

```
Options +FollowSymLinks
IndexIgnore */*

RewriteEngine on

# if a directory or a file exists, use it directly
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d

# otherwise forward it to index.php
RewriteRule . index.php
```

In case of nginx you should not need any extra config files.

Check requirements

In order to run Yii, your webserver must meet its requirements. The very minimum requirement is PHP 5.4. In order to check the requirements copy `requirements.php` from your root directory into the webroot directory and run it via browser using `http://example.com/requirements.php` URL. Don't forget to delete the file afterwards.

Deploying an advanced project template

Deploying an advanced application to shared hosting is a bit trickier than a basic application but it could be achieved. Follow instructions described in advanced project template documentation.

Using template engines

By default, Yii uses PHP as its template language, but you can configure Yii to support other rendering engines, such as Twig or Smarty available as extensions.

The `view` component is responsible for rendering views. You can add a custom template engine by reconfiguring this component's behavior:

```
[
    'components' => [
        'view' => [
            'class' => 'yii\web\View',
            'renderers' => [
                'tpl' => [
                    'class' => 'yii\smarty\ViewRenderer',
                    //'cachePath' => '@runtime/Smarty/cache',
                ],
                'twig' => [
                    'class' => 'yii\twig\ViewRenderer',
                    'cachePath' => '@runtime/Twig/cache',
                    // Array of twig options:
                    'options' => [
                        'auto_reload' => true,
                    ],
                    'globals' => ['html' => '\yii\helpers\Html'],
                    'uses' => ['yii\bootstrap'],
                ],
                // ...
            ],
        ],
    ],
]
```

In the code above, both Smarty and Twig are configured to be useable by the view files. But in order to get these extensions into your project, you need to also modify your `composer.json` file to include them, too:

```
"yiisoft/yii2-smarty": "~2.0.0",
"yiisoft/yii2-twig": "~2.0.0",
```

That code would be added to the `require` section of `composer.json`. After making that change and saving the file, you can install the extensions by running `composer update --prefer-dist` in the command-line.

For details about using concrete template engine please refer to its documentation:

- Twig guide
- Smarty guide

Working with Third-Party Code

From time to time, you may need to use some third-party code in your Yii applications. Or you may want to use Yii as a library in some third-party systems. In this section, we will show how to achieve these goals.

Using Third-Party Libraries in Yii

To use a third-party library in a Yii application, you mainly need to make sure the classes in the library are properly included or can be autoloaded.

Using Composer Packages

Many third-party libraries are released in terms of Composer packages. You can install such libraries by taking the following two simple steps:

1. modify the `composer.json` file of your application and specify which Composer packages you want to install.
2. run `composer install` to install the specified packages.

The classes in the installed Composer packages can be autoloaded using the Composer autoloader. Make sure the entry script of your application contains the following lines to install the Composer autoloader:

```
// install Composer autoloader
require(__DIR__ . '/../vendor/autoload.php');

// include Yii class file
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');
```

Using Downloaded Libraries

If a library is not released as a Composer package, you should follow its installation instructions to install it. In most cases, you will need to download a release file manually and unpack it in the `BasePath/vendor` directory, where `BasePath` represents the base path of your application.

If a library carries its own class autoloader, you may install it in the entry script of your application. It is recommended the installation is done before you include the `Yii.php` file so that the Yii class autoloader can take precedence in autoloading classes.

If a library does not provide a class autoloader, but its class naming follows PSR-4, you may use the Yii class autoloader to autoload the classes. All you need to do is just to declare a root alias for each root namespace used in its classes. For example, assume you have installed a library in the directory `vendor/foo/bar`, and the library classes are under the `xyz` root namespace. You can include the following code in your application configuration:

```
[
    'aliases' => [
        '@xyz' => '@vendor/foo/bar',
    ],
]
```

If neither of the above is the case, it is likely that the library relies on PHP include path configuration to correctly locate and include class files. Simply follow its instruction on how to configure the PHP include path.

In the worst case when the library requires explicitly including every class file, you can use the following method to include the classes on demand:

- Identify which classes the library contains.
- List the classes and the corresponding file paths in `Yii::$classMap` in the entry script of the application. For example,

  ```
  Yii::$classMap['Class1'] = 'path/to/Class1.php';
  Yii::$classMap['Class2'] = 'path/to/Class2.php';
  ```

Using Yii in Third-Party Systems

Because Yii provides many excellent features, sometimes you may want to use some of its features to support developing or enhancing 3rd-party systems, such as WordPress, Joomla, or applications developed using other PHP frameworks. For example, you may want to use the [[yii\helpers\ArrayHelper]] class or use the Active Record feature in a third-party system. To achieve this goal, you mainly need to take two steps: install Yii, and bootstrap Yii.

If the third-party system uses Composer to manage its dependencies, you can simply run the following commands to install Yii:

```
composer global require "fxp/composer-asset-plugin:^1.2.0"
composer require yiisoft/yii2
composer install
```

The first command installs the composer asset plugin which allows managing bower and npm package dependencies through Composer. Even if you only want to use the database layer or other non-asset related features of Yii, this is required to install the Yii composer package.

If you want to use the Asset publishing feature of Yii you should also add the following configuration to the `extra` section in your `composer.json`:

```
{
    ...
    "extra": {
        "asset-installer-paths": {
            "npm-asset-library": "vendor/npm",
            "bower-asset-library": "vendor/bower"
        }
    }
}
```

See also the general section about installing Yii for more information on Composer and solution to possible issues popping up during the installation.

Otherwise, you can download the Yii release file and unpack it in the `BasePath/vendor` directory.

Next, you should modify the entry script of the 3rd-party system by including the following code at the beginning:

```
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

$yiiConfig = require(__DIR__ . '/../config/yii/web.php');
new yii\web\Application($yiiConfig); // Do NOT call run() here
```

As you can see, the code above is very similar to that in the entry script of a typical Yii application. The only difference is that after the application instance is created, the `run()` method is not called. This is because by calling `run()`, Yii will take over the control of the request handling workflow which is not needed in this case and already handled by the existing application.

Like in a Yii application, you should configure the application instance based on the environment running the third-party system. For example, to use the Active Record feature, you need to configure the `db` application component with the DB connection setting used by the third-party system.

Now you can use most features provided by Yii. For example, you can create Active Record classes and use them to work with databases.

Using Yii 2 with Yii 1

If you were using Yii 1 previously, it is likely you have a running Yii 1 application. Instead of rewriting the whole application in Yii 2, you may just want to enhance it using some of the features only available in Yii 2. This can be achieved as described below.

> Note: Yii 2 requires PHP 5.4 or above. You should make sure that both your server and the existing application support this.

First, install Yii 2 in your existing application by following the instructions given in the last subsection.

Second, modify the entry script of the application as follows,

```
// include the customized Yii class described below
require(__DIR__ . '/../components/Yii.php');

// configuration for Yii 2 application
$yii2Config = require(__DIR__ . '/../config/yii2/web.php');
new yii\web\Application($yii2Config); // Do NOT call run(), yii2 app is only used as s
ervice locator

// configuration for Yii 1 application
$yii1Config = require(__DIR__ . '/../config/yii1/main.php');
Yii::createWebApplication($yii1Config)->run();
```

Because both Yii 1 and Yii 2 have the `Yii` class, you should create a customized version to combine them. The above code includes the customized `Yii` class file, which can be created as follows.

```
$yii2path = '/path/to/yii2';
require($yii2path . '/BaseYii.php'); // Yii 2.x

$yii1path = '/path/to/yii1';
require($yii1path . '/YiiBase.php'); // Yii 1.x

class Yii extends \yii\BaseYii
{
    // copy-paste the code from YiiBase (1.x) here
}

Yii::$classMap = include($yii2path . '/classes.php');
// register Yii 2 autoloader via Yii 1
Yii::registerAutoloader(['yii\BaseYii', 'autoload']);
// create the dependency injection container
Yii::$container = new yii\di\Container;
```

That's all! Now in any part of your code, you can use `Yii::$app` to access the Yii 2 application instance, while `Yii::app()` will give you the Yii 1 application instance:

```
echo get_class(Yii::app()); // outputs 'CWebApplication'
echo get_class(Yii::$app);  // outputs 'yii\web\Application'
```

Helpers

> Note: This section is under development.

Yii provides many classes that help simplify common coding tasks, such as string or array manipulations, HTML code generation, and so on. These helper classes are organized under the `yii\helpers` namespace and are all static classes (meaning they contain only static properties and methods and should not be instantiated).

You use a helper class by directly calling one of its static methods, like the following:

```
use yii\helpers\Html;

echo Html::encode('Test > test');
```

> Note: To support customizing helper classes, Yii breaks each core helper class into two classes: a base class (e.g. `BaseArrayHelper`) and a concrete class (e.g. `ArrayHelper`). When you use a helper, you should only use the concrete version and never use the base class.

Core Helper Classes

The following core helper classes are provided in the Yii releases:

- ArrayHelper
- Console
- FileHelper
- FormatConverter
- Html
- HtmlPurifier
- Imagine (provided by yii2-imagine extension)
- Inflector
- Json
- Markdown
- StringHelper
- Url
- VarDumper

Customizing Helper Classes

To customize a core helper class (e.g. [[yii\helpers\ArrayHelper]]), you should create a new class extending from the helpers corresponding base class (e.g. [[yii\helpers\BaseArrayHelper]]) and name your class the same as the corresponding concrete class (e.g. [[yii\helpers\ArrayHelper]]), including its namespace. This class will then be set up to replace the original implementation of the framework.

The following example shows how to customize the [[yii\helpers\ArrayHelper::merge()|merge()]] method of the [[yii\helpers\ArrayHelper]] class:

```
<?php

namespace yii\helpers;

class ArrayHelper extends BaseArrayHelper
{
    public static function merge($a, $b)
    {
        // your custom implementation
    }
}
```

Save your class in a file named `ArrayHelper.php`. The file can be in any directory, for example `@app/components`.

Next, in your application's entry script, add the following line of code after including the `yii.php` file to tell the Yii class autoloader to load your custom class instead of the original helper class from the framework:

```
Yii::$classMap['yii\helpers\ArrayHelper'] = '@app/components/ArrayHelper.php';
```

Note that customizing of helper classes is only useful if you want to change the behavior of an existing function of the helpers. If you want to add additional functions to use in your application you may better create a separate helper for that.

ArrayHelper

Additionally to the rich set of PHP array functions, the Yii array helper provides extra static methods allowing you to deal with arrays more efficiently.

Getting Values

Retrieving values from an array, an object or a complex structure consisting of both using standard PHP is quite repetitive. You have to check if key exists with `isset` first, then if it does you're getting it, if not, providing default value:

```
class User
{
    public $name = 'Alex';
}

$array = [
    'foo' => [
        'bar' => new User(),
    ]
];

$value = isset($array['foo']['bar']->name) ? $array['foo']['bar']->name : null;
```

Yii provides a very convenient method to do it:

```
$value = ArrayHelper::getValue($array, 'foo.bar.name');
```

First method argument is where we're getting value from. Second argument specifies how to get the data. It could be one of the following:

- Name of array key or object property to retrieve value from.
- Set of dot separated array keys or object property names. The one we've used in the example above.
- A callback returning a value.

The callback should be the following:

```
$fullName = ArrayHelper::getValue($user, function ($user, $defaultValue) {
    return $user->firstName . ' ' . $user->lastName;
});
```

Third optional argument is default value which is `null` if not specified. Could be used as follows:

```
$username = ArrayHelper::getValue($comment, 'user.username', 'Unknown');
```

In case you want to get the value and then immediately remove it from array you can use `remove` method:

```
$array = ['type' => 'A', 'options' => [1, 2]];
$type = ArrayHelper::remove($array, 'type');
```

After executing the code `$array` will contain `['options' => [1, 2]]` and `$type` will be `A`. Note that unlike `getValue` method, `remove` supports simple key names only.

Checking Existence of Keys

`ArrayHelper::keyExists` works the same way as array_key_exists except that it also supports case-insensitive key comparison. For example,

```
$data1 = [
    'userName' => 'Alex',
];

$data2 = [
    'username' => 'Carsten',
];

if (!ArrayHelper::keyExists('username', $data1, false) || !ArrayHelper::keyExists('use
rname', $data2, false)) {
    echo "Please provide username.";
}
```

Retrieving Columns

Often you need to get a column of values from array of data rows or objects. Common example is getting a list of IDs.

```
$array = [
    ['id' => '123', 'data' => 'abc'],
    ['id' => '345', 'data' => 'def'],
];
$ids = ArrayHelper::getColumn($array, 'id');
```

The result will be `['123', '345']`.

If additional transformations are required or the way of getting value is complex, second argument could be specified as an anonymous function:

```
$result = ArrayHelper::getColumn($array, function ($element) {
    return $element['id'];
});
```

Re-indexing Arrays

In order to index an array according to a specified key, the `index` method can be used. The input should be either multidimensional array or an array of objects. The `$key` can be either a key name of the sub-array, a property name of object, or an anonymous function that must return the value that will be used as a key.

The `$groups` attribute is an array of keys, that will be used to group the input array into one or more sub-arrays based on keys specified.

If the `$key` attribute or its value for the particular element is `null` and `$groups` is not defined, the array element will be discarded. Otherwise, if `$groups` is specified, array element will be added to the result array without any key.

For example:

```
$array = [
    ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
    ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
];
$result = ArrayHelper::index($array, 'id');
```

The result will be an associative array, where the key is the value of `id` attribute:

```
[
    '123' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop'],
    '345' => ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone']
    // The second element of an original array is overwritten by the last element beca
use of the same id
]
```

Anonymous function, passed as a `$key`, gives the same result:

```
$result = ArrayHelper::index($array, function ($element) {
    return $element['id'];
});
```

Passing `id` as a third argument will group `$array` by `id`:

```
$result = ArrayHelper::index($array, null, 'id');
```

The result will be a multidimensional array grouped by `id` on the first level and not indexed on the second level:

```
[
    '123' => [
        ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
    ],
    '345' => [ // all elements with this index are present in the result array
        ['id' => '345', 'data' => 'def', 'device' => 'tablet'],
        ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone'],
    ]
]
```

An anonymous function can be used in the grouping array as well:

```
$result = ArrayHelper::index($array, 'data', [function ($element) {
    return $element['id'];
}, 'device']);
```

The result will be a multidimensional array grouped by `id` on the first level, by `device` on the second level and indexed by `data` on the third level:

```
[
    '123' => [
        'laptop' => [
            'abc' => ['id' => '123', 'data' => 'abc', 'device' => 'laptop']
        ]
    ],
    '345' => [
        'tablet' => [
            'def' => ['id' => '345', 'data' => 'def', 'device' => 'tablet']
        ],
        'smartphone' => [
            'hgi' => ['id' => '345', 'data' => 'hgi', 'device' => 'smartphone']
        ]
    ]
]
```

Building Maps

In order to build a map (key-value pairs) from a multidimensional array or an array of objects you can use `map` method. The `$from` and `$to` parameters specify the key names or property names to set up the map. Optionally, one can further group the map according to a grouping field `$group`. For example,

```
$array = [
    ['id' => '123', 'name' => 'aaa', 'class' => 'x'],
    ['id' => '124', 'name' => 'bbb', 'class' => 'x'],
    ['id' => '345', 'name' => 'ccc', 'class' => 'y'],
];

$result = ArrayHelper::map($array, 'id', 'name');
// the result is:
// [
//     '123' => 'aaa',
//     '124' => 'bbb',
//     '345' => 'ccc',
// ]

$result = ArrayHelper::map($array, 'id', 'name', 'class');
// the result is:
// [
//     'x' => [
//         '123' => 'aaa',
//         '124' => 'bbb',
//     ],
//     'y' => [
//         '345' => 'ccc',
//     ],
// ]
```

Multidimensional Sorting

`multisort` method helps to sort an array of objects or nested arrays by one or several keys. For example,

```
$data = [
    ['age' => 30, 'name' => 'Alexander'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 19, 'name' => 'Barney'],
];
ArrayHelper::multisort($data, ['age', 'name'], [SORT_ASC, SORT_DESC]);
```

After sorting we'll get the following in `$data`:

```
[
    ['age' => 19, 'name' => 'Barney'],
    ['age' => 30, 'name' => 'Brian'],
    ['age' => 30, 'name' => 'Alexander'],
];
```

Second argument that specifies keys to sort by can be a string if it's a single key, an array in case of multiple keys or an anonymous function like the following one:

```
ArrayHelper::multisort($data, function($item) {
    return isset($item['age']) ? ['age', 'name'] : 'name';
});
```

Third argument is direction. In case of sorting by a single key it could be either `SORT_ASC` or `SORT_DESC`. If sorting by multiple values you can sort each value differently by providing an array of sort direction.

Last argument is PHP sort flag that could take the same values as the ones passed to PHP sort().

Detecting Array Types

It is handy to know whether an array is indexed or an associative. Here's an example:

```
// no keys specified
$indexed = ['Qiang', 'Paul'];
echo ArrayHelper::isIndexed($indexed);

// all keys are strings
$associative = ['framework' => 'Yii', 'version' => '2.0'];
echo ArrayHelper::isAssociative($associative);
```

HTML Encoding and Decoding Values

In order to encode or decode special characters in an array of strings into HTML entities you can use the following:

```
$encoded = ArrayHelper::htmlEncode($data);
$decoded = ArrayHelper::htmlDecode($data);
```

Only values will be encoded by default. By passing second argument as `false` you can encode array's keys as well. Encoding will use application charset and could be changed via third argument.

Merging Arrays

You can use [[yii\helpers\ArrayHelper::merge()|ArrayHelper::merge()]] to merge two or more arrays into one recursively. If each array has an element with the same string key value, the latter will overwrite the former (different from array_merge_recursive()). Recursive merging will be conducted if both arrays have an element of array type and are having the same key. For integer-keyed elements, the elements from the latter array will be appended to the former array. You can use [[yii\helpers\UnsetArrayValue]] object to unset value from previous array or [[yii\helpers\ReplaceArrayValue]] to force replace former value instead of recursive merging.

For example:

```
$array1 = [
    'name' => 'Yii',
    'version' => '1.1',
    'ids' => [
        1,
    ],
    'validDomains' => [
        'example.com',
        'www.example.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
        'dev' => 'dev@example.com',
    ],
];

$array2 = [
    'version' => '2.0',
    'ids' => [
        2,
    ],
    'validDomains' => new \yii\helpers\ReplaceArrayValue([
        'yiiframework.com',
        'www.yiiframework.com',
    ]),
    'emails' => [
        'dev' => new \yii\helpers\UnsetArrayValue(),
    ],
];

$result = ArrayHelper::merge($array1, $array2);
```

The result will be:

```
[
    'name' => 'Yii',
    'version' => '2.0',
    'ids' => [
        1,
        2,
    ],
    'validDomains' => [
        'yiiframework.com',
        'www.yiiframework.com',
    ],
    'emails' => [
        'admin' => 'admin@example.com',
    ],
]
```

Converting Objects to Arrays

Often you need to convert an object or an array of objects into an array. The most common case is converting active record models in order to serve data arrays via REST API or use it otherwise. The following code could be used to do it:

```
$posts = Post::find()->limit(10)->all();
$data = ArrayHelper::toArray($posts, [
    'app\models\Post' => [
        'id',
        'title',
        // the key name in array result => property name
        'createTime' => 'created_at',
        // the key name in array result => anonymous function
        'length' => function ($post) {
            return strlen($post->content);
        },
    ],
]);
```

The first argument contains the data we want to convert. In our case we're converting a `Post` AR model.

The second argument is conversion mapping per class. We're setting a mapping for `Post` model. Each mapping array contains a set of mappings. Each mapping could be:

- A field name to include as is.
- A key-value pair of desired array key name and model column name to take value from.
- A key-value pair of desired array key name and a callback which returns value.

The result of conversion above for single model will be:

```
[
    'id' => 123,
    'title' => 'test',
    'createTime' => '2013-01-01 12:00AM',
    'length' => 301,
]
```

It is possible to provide default way of converting object to array for a specific class by implementing [[yii\base\Arrayable|Arrayable]] interface in that class.

Testing against Arrays

Often you need to check if an element is in an array or a set of elements is a subset of another. While PHP offers `in_array()`, this does not support subsets or `\Traversable` objects.

To aid these kinds of tests, [[yii\helpers\ArrayHelper]] provides [[yii\helpers\ArrayHelper::isIn()|isIn()]] and [[yii\helpers\ArrayHelper::isSubset()|isSubset()]] with the same signature as in_array().

```
// true
ArrayHelper::isIn('a', ['a']);
// true
ArrayHelper::isIn('a', new ArrayObject(['a']));

// true
ArrayHelper::isSubset(new ArrayObject(['a', 'c']), new ArrayObject(['a', 'b', 'c']));
```

Html helper

Every web application generates lots of HTML markup. If the markup is static, it can be done efficiently by mixing PHP and HTML in a single file, but when it is generated dynamically it starts to get tricky to handle it without extra help. Yii provides such help in the form of an Html helper, which provides a set of static methods for handling commonly used HTML tags, their options, and their content.

> Note: If your markup is nearly static, it's better to use HTML directly. There's no need to wrap absolutely everything in Html helper calls.

Basics

Since building dynamic HTML by string concatenation can get messy very fast, Yii provides a set of methods to manipulate tag options and build tags based on these options.

Generating Tags

The code for generating a tag looks like the following:

```
<?= Html::tag('p', Html::encode($user->name), ['class' => 'username']) ?>
```

The first argument is the tag name. The second one is the content to be enclosed between the start and end tags. Note that we are using `Html::encode` — that's because the content isn't encoded automatically to allow using HTML when needed. The third one is an array of HTML options, or in other words, tag attributes. In this array the key is the name of the attribute (such as `class`, `href` or `target`), and the value is its value.

The code above will generate the following HTML:

```
<p class="username">samdark</p>
```

In case you need just an opening or closing tag, you can use the `Html::beginTag()` and `Html::endTag()` methods.

Options are used in many methods of the Html helper and various widgets. In all these cases there is some extra handling to know about:

- If a value is `null`, the corresponding attribute will not be rendered.

- Attributes whose values are of boolean type will be treated as boolean attributes.
- The values of attributes will be HTML-encoded using [[yii\helpers\Html::encode()|Html::encode()]].
- If the value of an attribute is an array, it will be handled as follows:
  - If the attribute is a data attribute as listed in [[yii\helpers\Html::$dataAttributes]], such as `data` or `ng`, a list of attributes will be rendered, one for each element in the value array. For example, `'data' => ['id' => 1, 'name' => 'yii']` generates `data-id="1" data-name="yii"`; and `'data' => ['params' => ['id' => 1, 'name' => 'yii'], 'status' => 'ok']` generates `data-params='{"id":1,"name":"yii"}' data-status="ok"`. Note that in the latter example JSON format is used to render a sub-array.
  - If the attribute is NOT a data attribute, the value will be JSON-encoded. For example, `['params' => ['id' => 1, 'name' => 'yii']` generates `params='{"id":1,"name":"yii"}'`.

Forming CSS Classes and Styles

When building options for HTML tags we often start with defaults which we need to modify. In order to add or remove a CSS class you can use the following:

```
$options = ['class' => 'btn btn-default'];

if ($type === 'success') {
    Html::removeCssClass($options, 'btn-default');
    Html::addCssClass($options, 'btn-success');
}

echo Html::tag('div', 'Pwede na', $options);

// if the value of $type is 'success' it will render
// <div class="btn btn-success">Pwede na</div>
```

You may specify multiple CSS classes using the array style as well:

```
$options = ['class' => ['btn', 'btn-default']];

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-default">Save</div>'
```

You may also use the array style when adding or removing classes:

```
$options = ['class' => 'btn'];

if ($type === 'success') {
    Html::addCssClass($options, ['btn-success', 'btn-lg']);
}

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-success btn-lg">Save</div>'
```

`Html::addCssClass()` prevents duplication, so you don't need to worry about the same class appearing twice:

```
$options = ['class' => 'btn btn-default'];

Html::addCssClass($options, 'btn-default'); // class 'btn-default' is already present

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-default">Save</div>'
```

If the CSS class option is specified using the array style, you may use a named key to mark the logical purpose of the class. In this case, a class with the same key in the array style will be ignored in `Html::addCssClass()`:

```
$options = [
    'class' => [
        'btn',
        'theme' => 'btn-default',
    ]
];

Html::addCssClass($options, ['theme' => 'btn-success']); // 'theme' key is already tak
en

echo Html::tag('div', 'Save', $options);
// renders '<div class="btn btn-default">Save</div>'
```

CSS styles can be set up in similar way using the `style` attribute:

```
$options = ['style' => ['width' => '100px', 'height' => '100px']];

// gives style="width: 100px; height: 200px; position: absolute;"
Html::addCssStyle($options, 'height: 200px; position: absolute;');

// gives style="position: absolute;"
Html::removeCssStyle($options, ['width', 'height']);
```

When using [[yii\helpers\Html::addCssStyle()|addCssStyle()]], you can specify either an array of key-value pairs, corresponding to CSS property names and values, or a string such as `width: 100px; height: 200px;`. These formats can be converted from one to the other using [[yii\helpers\Html::cssStyleFromArray()|cssStyleFromArray()]] and [[yii\helpers\Html::cssStyleToArray()|cssStyleToArray()]]. The [[yii\helpers\Html::removeCssStyle()|removeCssStyle()]] method accepts an array of properties to remove. If it's a single property, it can be specified as a string.

Encoding and Decoding Content

In order for content to be displayed properly and securely in HTML, special characters in the content should be encoded. In PHP this is done with htmlspecialchars and htmlspecialchars_decode. The issue with using these methods directly is that you have to specify encoding and extra flags all the time. Since these flags are the same all the time and the encoding should match the one of the application in order to prevent security issues, Yii provides two compact and simple-to-use methods:

```
$userName = Html::encode($user->name);
echo $userName;

$decodedUserName = Html::decode($userName);
```

Forms

Dealing with form markup is quite repetitive and error prone. Because of that, there is a group of methods to help dealing with them.

> Note: consider using [[yii\widgets\ActiveForm|ActiveForm]] in case you're dealing with models and need validation.

Creating Forms

Forms can be opened with [[yii\helpers\Html::beginForm()|beginForm()]] method like the following:

```
<?= Html::beginForm(['order/update', 'id' => $id], 'post', ['enctype' => 'multipart/fo
rm-data']) ?>
```

The first argument is the URL the form will be submitted to. It can be specified in the form of a Yii route and parameters accepted by [[yii\helpers\Url::to()|Url::to()]]. The second one is the method to use. `post` is the default. The third one is an array of options for the form tag. In

this case we're changing the encoding of the form data in the POST request to `multipart/form-data`, which is required in order to upload files.

Closing the form tag is simple:

```
<?= Html::endForm() ?>
```

Buttons

In order to generate buttons, you can use the following code:

```
<?= Html::button('Press me!', ['class' => 'teaser']) ?>
<?= Html::submitButton('Submit', ['class' => 'submit']) ?>
<?= Html::resetButton('Reset', ['class' => 'reset']) ?>
```

The first argument for all three methods is the button title, and the second one is an array of options. The title isn't encoded, so if you're displaying data from the end user, encode it with [[yii\helpers\Html::encode()|Html::encode()]].

Input Fields

There are two groups of input methods. The ones starting with `active`, which are called active inputs, and the ones not starting with it. Active inputs take data from the model and attribute specified, while in the case of a regular input, data is specified directly.

The most generic methods are:

```
type, input name, input value, options
<?= Html::input('text', 'username', $user->name, ['class' => $username]) ?>

type, model, model attribute name, options
<?= Html::activeInput('text', $user, 'name', ['class' => $username]) ?>
```

If you know the input type in advance, it's more convenient to use the shortcut methods:

- [[yii\helpers\Html::buttonInput()]]
- [[yii\helpers\Html::submitInput()]]
- [[yii\helpers\Html::resetInput()]]
- [[yii\helpers\Html::textInput()]], [[yii\helpers\Html::activeTextInput()]]
- [[yii\helpers\Html::hiddenInput()]], [[yii\helpers\Html::activeHiddenInput()]]
- [[yii\helpers\Html::passwordInput()]] / [[yii\helpers\Html::activePasswordInput()]]
- [[yii\helpers\Html::fileInput()]], [[yii\helpers\Html::activeFileInput()]]
- [[yii\helpers\Html::textarea()]], [[yii\helpers\Html::activeTextarea()]]

Radios and checkboxes are a bit different in terms of method signature:

```
<?= Html::radio('agree', true, ['label' => 'I agree']);
<?= Html::activeRadio($model, 'agree', ['class' => 'agreement'])

<?= Html::checkbox('agree', true, ['label' => 'I agree']);
<?= Html::activeCheckbox($model, 'agree', ['class' => 'agreement'])
```

Dropdown lists and list boxes can be rendered like the following:

```
<?= Html::dropDownList('list', $currentUserId, ArrayHelper::map($userModels, 'id', 'na
me')) ?>
<?= Html::activeDropDownList($users, 'id', ArrayHelper::map($userModels, 'id', 'name')
) ?>

<?= Html::listBox('list', $currentUserId, ArrayHelper::map($userModels, 'id', 'name'))
?>
<?= Html::activeListBox($users, 'id', ArrayHelper::map($userModels, 'id', 'name')) ?>
```

The first argument is the name of the input, the second one is the value that's currently selected, and the third one is an array of key-value pairs, where the array key is the list value and the array value is the list label.

If you want multiple choices to be selectable, you can use a checkbox list:

```
<?= Html::checkboxList('roles', [16, 42], ArrayHelper::map($roleModels, 'id', 'name'))
?>
<?= Html::activeCheckboxList($user, 'role', ArrayHelper::map($roleModels, 'id', 'name'
)) ?>
```

If not, use radio list:

```
<?= Html::radioList('roles', [16, 42], ArrayHelper::map($roleModels, 'id', 'name')) ?>
<?= Html::activeRadioList($user, 'role', ArrayHelper::map($roleModels, 'id', 'name'))
?>
```

Labels and Errors

Same as inputs, there are two methods for generating form labels. Active, which takes data from the model, and non-active, which accepts data directly:

```
<?= Html::label('User name', 'username', ['class' => 'label username']) ?>
<?= Html::activeLabel($user, 'username', ['class' => 'label username']) ?>
```

In order to display form errors from a model or models as a summary, you could use:

```
<?= Html::errorSummary($posts, ['class' => 'errors']) ?>
```

To display an individual error:

```
<?= Html::error($post, 'title', ['class' => 'error']) ?>
```

Input Names and Values

There are methods to get names, ids and values for input fields based on the model. These are mainly used internally, but could be handy sometimes:

```
// Post[title]
echo Html::getInputName($post, 'title');

// post-title
echo Html::getInputId($post, 'title');

// my first post
echo Html::getAttributeValue($post, 'title');

// $post->authors[0]
echo Html::getAttributeValue($post, '[0]authors[0]');
```

In the above, the first argument is the model, while the second one is the attribute expression. In its simplest form the expression is just an attribute name, but it can be an attribute name prefixed and/or suffixed with array indexes, which is mainly used for tabular input:

- `[0]content` is used in tabular data input to represent the `content` attribute for the first model in tabular input;
- `dates[0]` represents the first array element of the `dates` attribute;
- `[0]dates[0]` represents the first array element of the `dates` attribute for the first model in tabular input.

In order to get the attribute name without suffixes or prefixes, one can use the following:

```
// dates
echo Html::getAttributeName('dates[0]');
```

Styles and Scripts

There are two methods to generate tags wrapping embedded styles and scripts:

```
<?= Html::style('.danger { color: #f00; }') ?>

Gives you

<style>.danger { color: #f00; }</style>


<?= Html::script('alert("Hello!");', ['defer' => true]);

Gives you

<script defer>alert("Hello!");</script>
```

If you want to use an external style in a CSS file:

```
<?= Html::cssFile('@web/css/ie5.css', ['condition' => 'IE 5']) ?>

generates

<!--[if IE 5]>
    <link href="http://example.com/css/ie5.css" />
<![endif]-->
```

The first argument is the URL. The second one is an array of options. In addition to the regular options, you can specify:

- `condition` to wrap `<link` in conditional comments with the specified condition. Hope you won't need conditional comments ever ;)
- `noscript` can be set to `true` to wrap `<link` in a `<noscript>` tag so it will be included only when there's either no JavaScript support in the browser or it was disabled by the user.

To link a JavaScript file:

```
<?= Html::jsFile('@web/js/main.js') ?>
```

Same as with CSS, the first argument specifies the URL of the file to be included. Options can be passed as the second argument. In the options you can specify `condition` in the same way as in the options for `cssFile`.

Hyperlinks

There's a method to generate hyperlinks conveniently:

```
<?= Html::a('Profile', ['user/view', 'id' => $id], ['class' => 'profile-link']) ?>
```

The first argument is the title. It's not encoded, so if you're using data entered by the user, you need to encode it with `Html::encode()`. The second argument is what will be in the `href` attribute of the `<a>` tag. See Url::to() for details on what values it accepts. The third argument is an array of tag attributes.

If you need to generate `mailto` links, you can use the following code:

```
<?= Html::mailto('Contact us', 'admin@example.com') ?>
```

Images

In order to generate an image tag, use the following:

```
<?= Html::img('@web/images/logo.png', ['alt' => 'My logo']) ?>

generates

<img src="http://example.com/images/logo.png" alt="My logo" />
```

Besides aliases, the first argument can accept routes, parameters and URLs, in the same way Url::to() does.

Lists

Unordered list can be generated like the following:

```
<?= Html::ul($posts, ['item' => function($item, $index) {
    return Html::tag(
        'li',
        $this->render('post', ['item' => $item]),
        ['class' => 'post']
    );
}]) ?>
```

In order to get ordered list, use `Html::ol()` instead.

Url Helper

Url helper provides a set of static methods for managing URLs.

Getting Common URLs

There are two methods you can use to get common URLs: home URL and base URL of the current request. In order to get home URL, use the following:

```
$relativeHomeUrl = Url::home();
$absoluteHomeUrl = Url::home(true);
$httpsAbsoluteHomeUrl = Url::home('https');
```

If no parameter is passed, the generated URL is relative. You can either pass `true` to get an absolute URL for the current schema or specify a schema explicitly (`https` , `http`).

To get the base URL of the current request use the following:

```
$relativeBaseUrl = Url::base();
$absoluteBaseUrl = Url::base(true);
$httpsAbsoluteBaseUrl = Url::base('https');
```

The only parameter of the method works exactly the same as for `Url::home()` .

Creating URLs

In order to create a URL to a given route use the `Url::toRoute()` method. The method uses [[\yii\web\UrlManager]] to create a URL:

```
$url = Url::toRoute(['product/view', 'id' => 42]);
```

You may specify the route as a string, e.g., `site/index` . You may also use an array if you want to specify additional query parameters for the URL being created. The array format must be:

```
// generates: /index.php?r=site%2Findex&param1=value1&param2=value2
['site/index', 'param1' => 'value1', 'param2' => 'value2']
```

If you want to create a URL with an anchor, you can use the array format with a `#` parameter. For example,

```
// generates: /index.php?r=site%2Findex&param1=value1#name
['site/index', 'param1' => 'value1', '#' => 'name']
```

A route may be either absolute or relative. An absolute route has a leading slash (e.g. `/site/index`) while a relative route has none (e.g. `site/index` or `index`). A relative route will be converted into an absolute one by the following rules:

- If the route is an empty string, the current [[\yii\web\Controller::route|route]] will be used;
- If the route contains no slashes at all (e.g. `index`), it is considered to be an action ID of the current controller and will be prepended with [[\yii\web\Controller::uniqueId]];
- If the route has no leading slash (e.g. `site/index`), it is considered to be a route relative to the current module and will be prepended with the module's [[\yii\base\Module::uniqueId|uniqueId]].

Starting from version 2.0.2, you may specify a route in terms of an alias. If this is the case, the alias will first be converted into the actual route which will then be turned into an absolute route according to the above rules.

Below are some examples of using this method:

```
// /index.php?r=site%2Findex
echo Url::toRoute('site/index');

// /index.php?r=site%2Findex&src=ref1#name
echo Url::toRoute(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post%2Fedit&id=100     assume the alias "@postEdit" is defined as "pos
t/edit"
echo Url::toRoute(['@postEdit', 'id' => 100]);

// http://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', true);

// https://www.example.com/index.php?r=site%2Findex
echo Url::toRoute('site/index', 'https');
```

There's another method `Url::to()` that is very similar to [[toRoute()]]. The only difference is that this method requires a route to be specified as an array only. If a string is given, it will be treated as a URL.

The first argument could be:

- an array: [[toRoute()]] will be called to generate the URL. For example: `['site/index']`,

`['post/index', 'page' => 2]`. Please refer to [[toRoute()]] for more details on how to specify a route.

- a string with a leading `@`: it is treated as an alias, and the corresponding aliased string will be returned.
- an empty string: the currently requested URL will be returned;
- a normal string: it will be returned as is.

When `$scheme` is specified (either a string or `true`), an absolute URL with host info (obtained from [[\yii\web\UrlManager::hostInfo]]) will be returned. If `$url` is already an absolute URL, its scheme will be replaced with the specified one.

Below are some usage examples:

```
// /index.php?r=site%2Findex
echo Url::to(['site/index']);

// /index.php?r=site%2Findex&src=ref1#name
echo Url::to(['site/index', 'src' => 'ref1', '#' => 'name']);

// /index.php?r=post%2Fedit&id=100     assume the alias "@postEdit" is defined as "pos
t/edit"
echo Url::to(['@postEdit', 'id' => 100]);

// the currently requested URL
echo Url::to();

// /images/logo.gif
echo Url::to('@web/images/logo.gif');

// images/logo.gif
echo Url::to('images/logo.gif');

// http://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', true);

// https://www.example.com/images/logo.gif
echo Url::to('@web/images/logo.gif', 'https');
```

Starting from version 2.0.3, you may use [[yii\helpers\Url::current()]] to create a URL based on the currently requested route and GET parameters. You may modify or remove some of the GET parameters or add new ones by passing a `$params` parameter to the method. For example,

```
// assume $_GET = ['id' => 123, 'src' => 'google'], current route is "post/view"

// /index.php?r=post%2Fview&id=123&src=google
echo Url::current();

// /index.php?r=post%2Fview&id=123
echo Url::current(['src' => null]);
// /index.php?r=post%2Fview&id=100&src=google
echo Url::current(['id' => 100]);
```

Remember URLs

There are cases when you need to remember URL and afterwards use it during processing of the one of sequential requests. It can be achieved in the following way:

```
// Remember current URL
Url::remember();

// Remember URL specified. See Url::to() for argument format.
Url::remember(['product/view', 'id' => 42]);

// Remember URL specified with a name given
Url::remember(['product/view', 'id' => 42], 'product');
```

In the next request we can get URL remembered in the following way:

```
$url = Url::previous();
$productUrl = Url::previous('product');
```

Checking Relative URLs

To find out if URL is relative i.e. it doesn't have host info part, you can use the following code:

```
$isRelative = Url::isRelative('test/it');
```