The Definitive Guide to Yii 2.0

Qiang Xue,
Alexander Makarov,
Carsten Brandt,
Klimov Paul
and
the Yii community

Contents

Chapter 1

Introduction

1.1 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**!

1.1.1 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 based on 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.

1.1.2 How does Yii Compare with Other Frameworks?

- Like most PHP frameworks, Yii implements the MVC (Model-View-Controller) design pattern and promotes code organization based on this pattern.

- Yii takes the philosophy that code should be written in a simple yet elegant way. It will never try to over-design things mainly for the purpose of following some design pattern.

- Yii is a full-stack framework providing many proven and ready-to-use features, such as: 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 core code. You can also take advantage of its 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[1] as well as a large community with many professionals constantly contributing to the development of Yii. The Yii developer team keeps a close eye on the latest trends of Web development, 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.

1.1.3 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 latest generation of the framework and will receive our main development efforts in the next few years. This guide is mainly about version 2.0.

1.1.4 Requirements and Prerequisites

Yii 2.0 requires PHP 5.4.0 or above. You can find more detailed requirements for individual features by running the requirement checker included in every Yii release.

Using Yii requires basic knowledge about 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[2] and traits[3]. Understanding these concepts will help you more easily pick up Yii 2.0.

1.2 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".

[1] `http://www.yiiframework.com/about/`
[2] `http://www.php.net/manual/en/language.namespaces.php`
[3] `http://www.php.net/manual/en/language.oop5.traits.php`

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.

1.2.1 Installation

Yii 2.0 fully embraces Composer[4], the de facto PHP package manager. Installation of the core framework, as well as extensions, are handled through Composer. Please refer to the Starting from Basic App 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.

1.2.2 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[5].
- Anonymous functions[6].
- 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[7].
- Late Static Bindings[8].
- Date and Time[9].
- Traits[10].
- intl[11]. Yii 2.0 makes use of the `intl` PHP extension to support internationalization features.

[4]https://getcomposer.org/
[5]http://php.net/manual/en/language.namespaces.php
[6]http://php.net/manual/en/functions.anonymous.php
[7]http://php.net/manual/en/book.spl.php
[8]http://php.net/manual/en/language.oop5.late-static-bindings.php
[9]http://php.net/manual/en/book.datetime.php
[10]http://php.net/manual/en/language.oop5.traits.php
[11]http://php.net/manual/en/book.intl.php

1.2.3 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.)

1.2.4 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` class is a lightweight base class that allows defining object properties via getters and setters. The `yii\base\Component` class extends from `yii\base\Object` and supports events and behaviors.

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

1.2.5 Object Configuration

The `yii\base\Object` class introduces a uniform way of configuring objects. Any descendant class of `yii\base\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()` 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 Object Configurations section.

1.2.6 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()` method:

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

To attach a handler to an event, use the `yii\base\Component::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.

1.2.7 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 the 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 Path Aliases section.

1.2.8 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. In 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` property. Please refer to the Template Engines section for more details.

1.2.9 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()` 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', '!name'],
    ];
}
```

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

In most cases, you do not need to override `yii\base\Model::scenarios()` if the `yii\base\Model::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.

1.2.10 Controllers

Yii 2.0 uses `yii\web\Controller` as the base controller class, similar to `CWebController` 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.

1.2.11 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()`, `yii\base\Widget::end()`, and `yii\base\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.

1.2.12 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.

1.2.13 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.

1.2.14 I18N

Yii 2.0 removes the built-in date formatter and number formatter pieces in favor of the PECL intl PHP module[12].

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.

1.2.15 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:

[12]`http://pecl.php.net/package/intl`

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

Please refer to the Filtering section for more details.

1.2.16 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.

1.2.17 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`
- `yii\helpers\Security`

Please refer to the Helper Overview section for more details.

1.2.18 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` 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.

1.2.19 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` object that can be turned into a SQL statement with the help of `yii\db\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.

1.2.20 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` 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` objects, you may chain the `yii\db\ActiveQuery::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 where 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.

1.2.21 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 application template provides such an example.

Please refer to the Authentication, Authorization, and Advanced Application Technique sections for more details.

1.2.22 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.

1.2.23 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.

Chapter 2

Getting Started

2.1 Installing Yii

You can install Yii in two ways, using Composer[1] 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.

> Note: Unlike with Yii 1, standard installations of Yii 2 results in both the framework and an application skeleton being downloaded and installed.

2.1.1 Installing via Composer

If you do not already have Composer installed, you may do so by following the instructions at getcomposer.org[2]. On Linux and Mac OS X, you'll run the following commands:

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

On Windows, you'll download and run Composer-Setup.exe[3].

Please refer to the Composer Documentation[4] if you encounter any problems or want to learn more about Composer usage.

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

```
composer create-project --prefer-dist yiisoft/yii2-app-basic basic
```

The above command installs Yii as a directory named `basic`.

[1] `http://getcomposer.org/`
[2] `https://getcomposer.org/download/`
[3] `https://getcomposer.org/Composer-Setup.exe`
[4] `https://getcomposer.org/doc/`

> Tip: If you want to install the latest development version of Yii, you may use the following command, which adds a stability option[5]:
>
> ```
> composer create-project --prefer-dist --stability=dev yiisoft/
>     yii2-app-basic basic
> ```
>
> Note that the development version of Yii should not be used for production as it may break your running code.

2.1.2 Installing from an Archive File

Installing Yii from an archive file involves two steps:

1. Download the archive file from yiiframework.com[6].
2. Unpack the downloaded file to a Web-accessible folder.

2.1.3 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 small projects, or for when 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 Application Template.

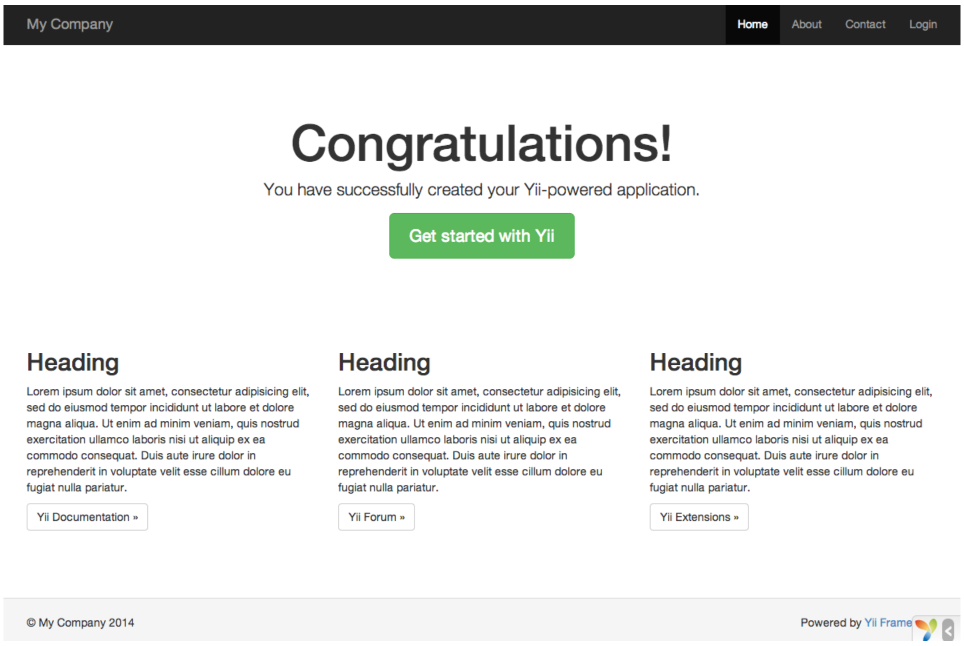
2.1.4 Verifying the Installation

After installation, you can use your browser to access the installed Yii application with the following URL:

```
http://localhost/basic/web/index.php
```

This URL assumes you have installed Yii in a directory named `basic`, directly under the Web server's document root directory, and that the Web server is running on your local machine(`localhost`), you may have to adjust it to your installation environment.

[5] https://getcomposer.org/doc/04-schema.md#minimum-stability
[6] http://www.yiiframework.com/download/yii2-basic


My Company

Home About Contact Login

Congratulations!

You have successfully created your Yii-powered application.

Get started with Yii

Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Yii Documentation »

Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Yii Forum »

Heading

Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat. Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu fugiat nulla pariatur.

Yii Extensions »

© My Company 2014

Powered by Yii Frame

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:

- Use a browser to access the URL `http://localhost/basic/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. You should also install the PDO PHP Extension[7] and a corresponding database driver (such as `pdo_mysql` for MySQL databases), if your application needs a database.

2.1.5 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.

[7] `http://www.php.net/manual/en/pdo.installation.php`

The application installed according to the above instructions should work out of box with either an Apache HTTP server[8] or an Nginx HTTP server[9], on Windows, Mac OS X, or Linux.

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 URL Parsing and Generation 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 producent 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">
    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>
```

[8]http://httpd.apache.org/
[9]http://nginx.org/

Recommended Nginx Configuration

You should have installed PHP as an FPM SAPI[10] to use Nginx[11]. 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 main;
    error_log   /path/to/basic/log/error.log;

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

    location ~ \.php$ {
        include fastcgi.conf;
        fastcgi_pass   127.0.0.1:9000;
        #fastcgi_pass unix:/var/run/php5-fpm.sock;
    }

    location ~ /\.(ht|svn|git) {
        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.

[10]http://php.net/install.fpm
[11]http://wiki.nginx.org/

2.2 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.

2.2.1 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.

2.2.2 Application Structure

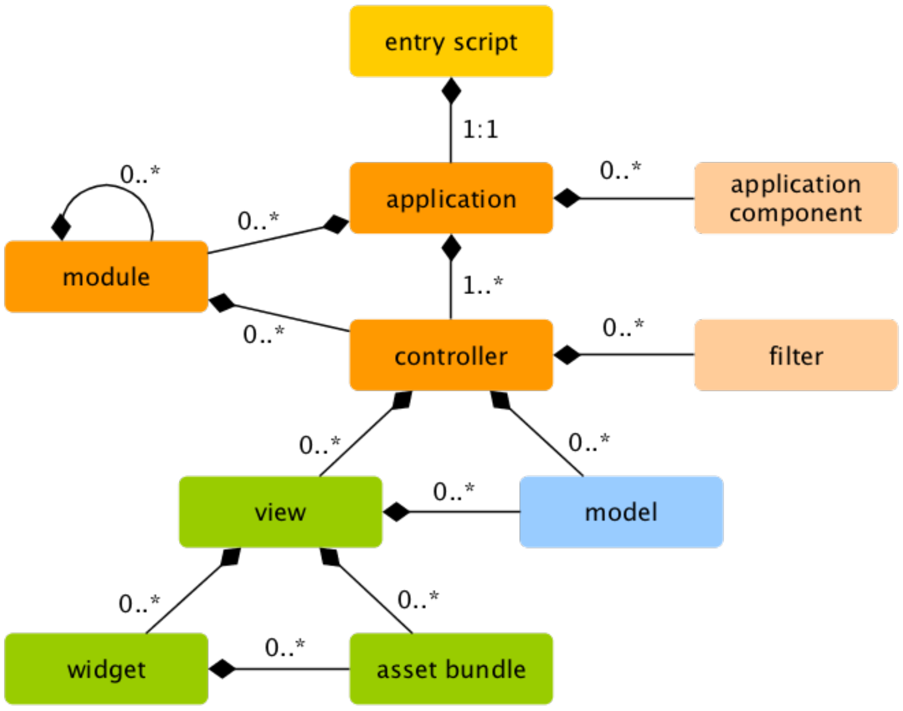
The most important directories and files in your application are (assuming the application's root directory is `basic`):

```
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 and cache files
    vendor/              contains the installed Composer packages, including
    the Yii framework 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)[12] design 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.

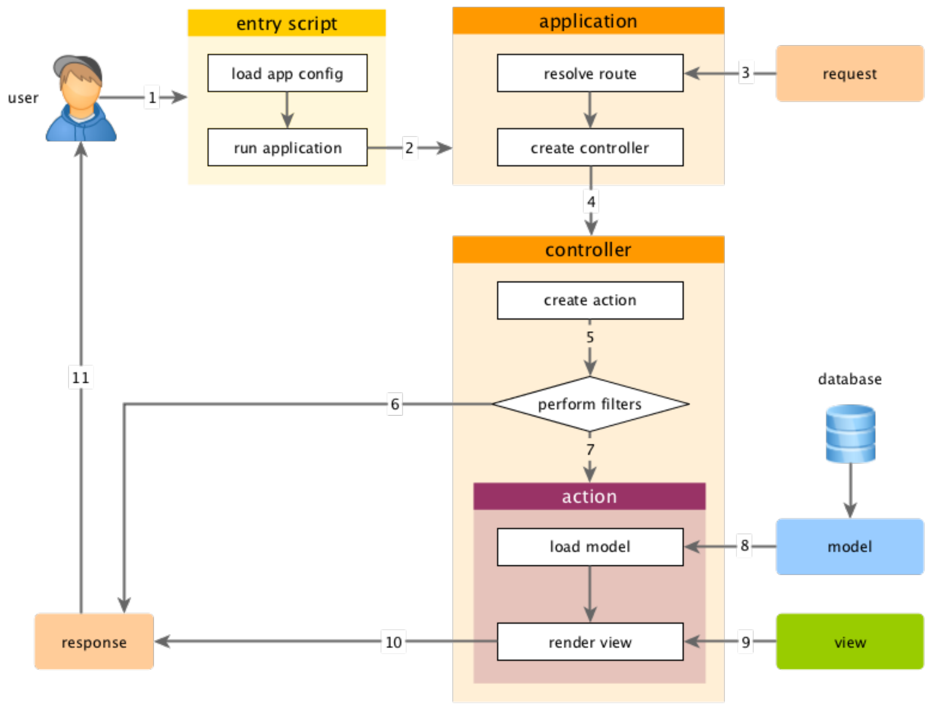


[12] `http://wikipedia.org/wiki/Model-view-controller`

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.

2.2.3 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.

2.3 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.

2.3.1 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 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()` 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).

2.3.2 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, and passed by the action to the view:

```
<?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()` is called in an action, it will look for a PHP file named as `views/ControllerID/ViewName.php`.

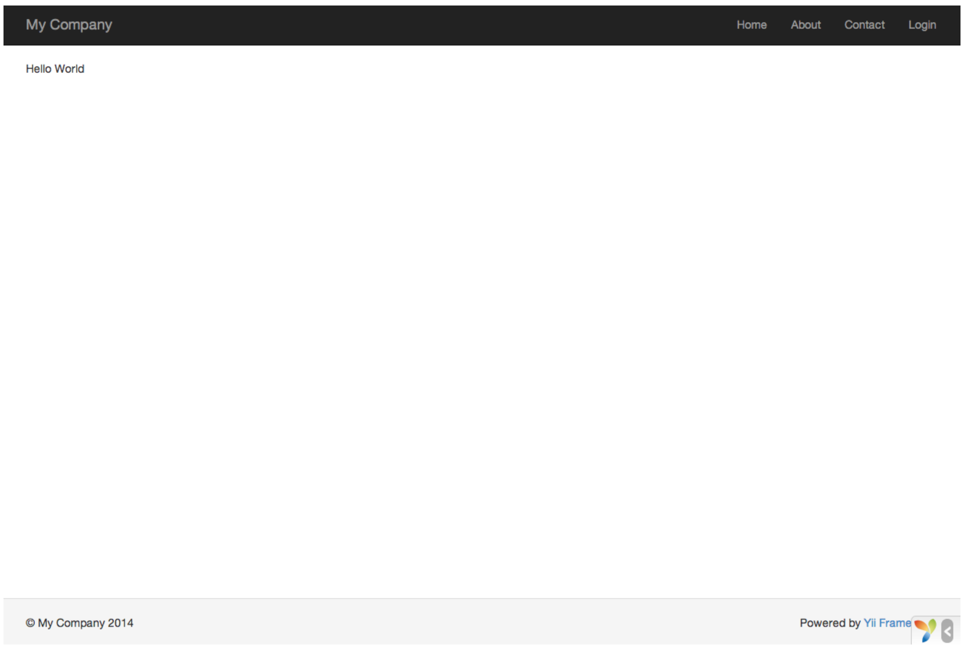
Note that in the above code, the `message` parameter is `yii\helpers\Html::encode()` before being printed. This is necessary as the parameter comes from an end user, making it vulnerable to cross-site scripting (XSS) attacks[13] by embedding malicious JavaScript code in the parameter.

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()` 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.

2.3.3 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/say&message=Hello+World
```



Hello World

© My Company 2014

Powered by Yii Frame

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

[13] `http://en.wikipedia.org/wiki/Cross-site_scripting`

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()` 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`.

2.3.4 Summary

In this section, you have touched the controller and view parts of the MVC design 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.

2.4 Working with Forms

In this section, we will describe how to create a new page to get data from users. The page will display a form with a name input field and an email input field. After getting these data from a user, the page will echo them back to the user 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;
- How to declare rules to validate the data entered by users;
- How to build an HTML form in a view.

2.4.1 Creating a Model

To represent the data entered by a user, create an `EntryForm` model class as shown below and save the class in the file `models/EntryForm.php`. Please refer to the Class Autoloading section for more details about the class file naming convention.

```php
<?php

namespace app\models;

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 that is commonly used to represent form data.

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

- both the `name` and `email` data are required;
- the `email` data must be a 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()` to trigger the data validation. A data validation failure will turn on the `yii\base\Model::hasErrors` property, and through `yii\base\Model::getErrors` you may learn what validation errors the model has.

2.4.2 Creating an Action

Next, create an `entry` action in the `site` controller, like you did in the previous 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` which is provided in Yii through `yii\web\Request::post()`. If the model is successfully populated (i.e., the user has submitted the HTML form), it will call `yii\base\Model::validate()` to make sure the data entered are valid.

If everything is fine, the action will render a view named `entry-confirm` to confirm with the user that the data he has entered is accepted. Otherwise, the `entry` view will be rendered, which will show the HTML form together with the validation error messages (if any).

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

2.4.3 Creating Views

Finally, create two views named `entry-confirm` and `entry` that are rendered by the `entry` action, as described in the last subsection.

The `entry-confirm` view simply displays the name and email data. It should be stored as 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 as 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(); ?>
```

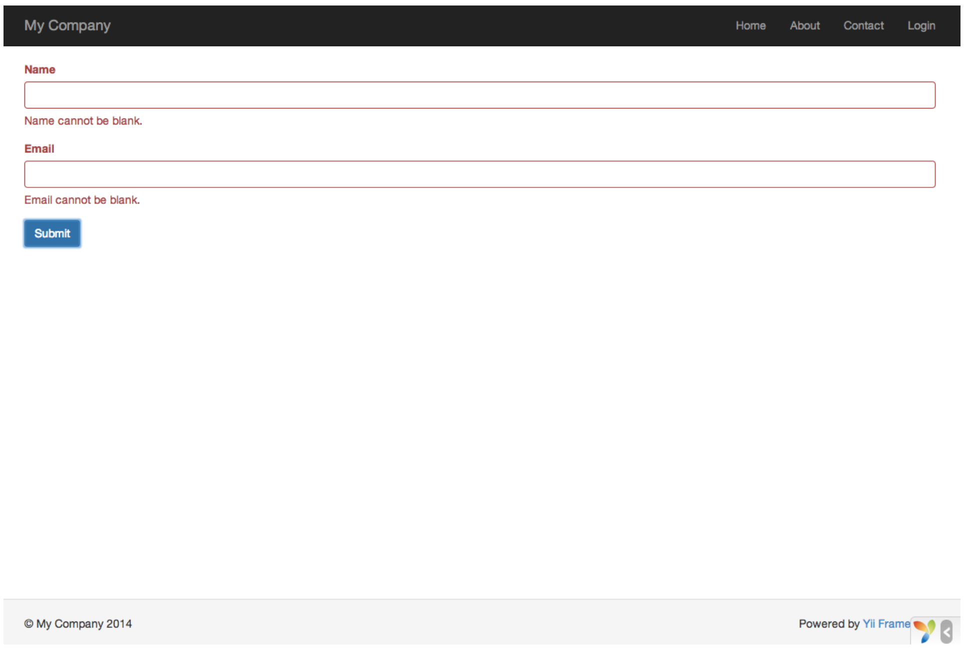
The view uses a powerful widget called `yii\widgets\ActiveForm` to build the HTML form. The `begin()` and `end()` methods of the widget render the opening and close form tags, respectively. Between the two method calls, input fields are created by the `yii\widgets\ActiveForm::field()` method. The first input field is about the “name” data, and the second the “email” data. After the input fields, the `yii\helpers\Html::submitButton()` method is called to generate a submit button.

2.4.4 Trying it Out

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

```
http://hostname/index.php?r=site/entry
```

You will see a page displaying a form with two input fields. In front of each input field, a label is also displayed indicating what data you need to enter. 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 that is displayed next to each problematic input field.



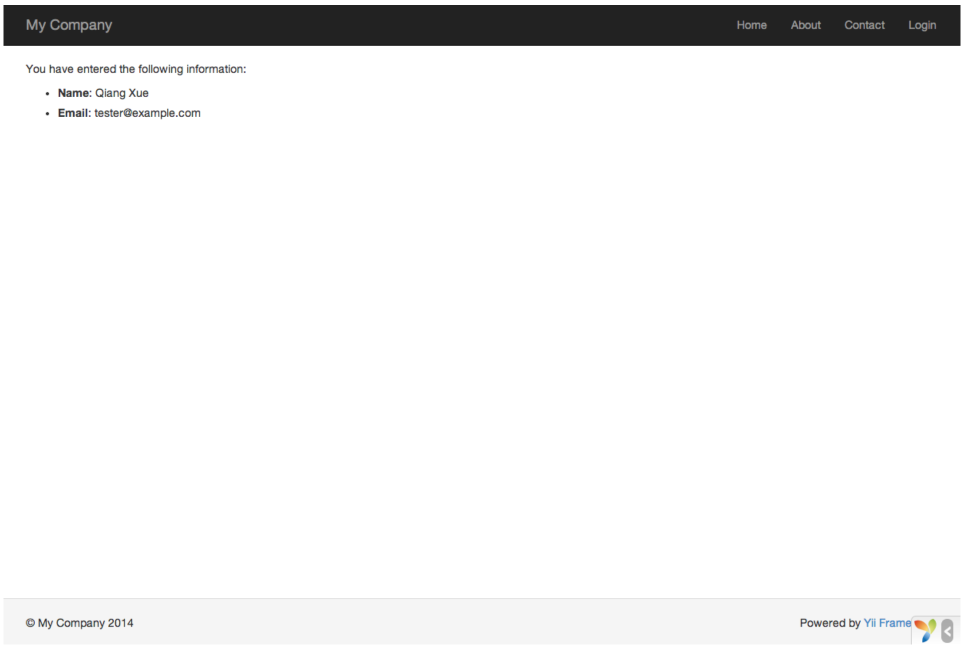
Name

Name cannot be blank.

Email

Email cannot be blank.

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.



You have entered the following information:

- **Name:** Qiang Xue
- **Email:** tester@example.com

© My Company 2014

Powered by Yii Frame

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 actually done on the client side using JavaScript as well as on the server side. `yii\widgets\ActiveForm` is smart enough to extract the validation rules that you have declared in `EntryForm`, turn them into 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.

The labels for input fields are generated by the `field()` method based on the model property names. For example, the label `Name` will be generated for the `name` property. You may customize a label by 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 turn much of your view code into reusable widgets to simplify view development in future.

2.4.5 Summary

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

You have also learned how to get data from users and how to display them back. This is a task that could take you a lot of time when developing an application. 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.

2.5 Working with Databases

In this section, we will describe how to create a new page to display the country data fetched from from a database table `country`. To achieve this goal, you will configure a database connection, create an Active Record class, and then create an action and a view.

Through this tutorial, you will learn

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

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

2.5.1 Preparing a Database

To begin with, create a database named `yii2basic` from which you will fetch data in your application. You may create a SQLite, MySQL, PostgreSQL, MSSQL or Oracle database. For simplicity, we will use MySQL in the following description.

Create a table named `country` in the database and insert some sample data. You may run the following SQL statements.

```
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',18886000);
INSERT INTO `Country` VALUES ('BR','Brazil',170115000);
INSERT INTO `Country` VALUES ('CA','Canada',1147000);
INSERT INTO `Country` VALUES ('CN','China',1277558000);
INSERT INTO `Country` VALUES ('DE','Germany',82164700);
INSERT INTO `Country` VALUES ('FR','France',59225700);
INSERT INTO `Country` VALUES ('GB','United Kingdom',59623400);
INSERT INTO `Country` VALUES ('IN','India',1013662000);
INSERT INTO `Country` VALUES ('RU','Russia',146934000);
INSERT INTO `Country` VALUES ('US','United States',278357000);
```

To this end, you have a database named `yii2basic`, and within this database there is a `country` table with ten rows of data.

2.5.2 Configuring a DB Connection

Make sure you have installed the PDO[14] 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.

Open the file `config/db.php` and adjust the content based on your database information. By default, the file contains the following content:

```
<?php

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

This is a typical file-based configuration. It 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 code via the expression `Yii::$app->db`.

> Info: The `config/db.php` file will be included in 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.

2.5.3 Creating an Active Record

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

```
<?php

namespace app\models;
```

[14] http://www.php.net/manual/en/book.pdo.php

```
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. Yii will guess the associated table name from the class name. In case this does not work, you may override the `yii\db\ActiveRecord::tableName()` method to explicitly specify the associated table name.

Using the `Country` class, you can manipulate the data in the `country` table easily. Below are some code snippets showing how you can make use of the `Country` class.

```
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 of accessing and manipulating database data in an object-oriented fashion. You may find more detailed information in the Active Record. Besides Active Record, you may also use a lower-level data accessing method called Data Access Objects.

2.5.4 Creating an Action

To expose the country data to end users, you need to create a new action. Instead of doing this in the `site` controller like you did in the previous sections, it makes more sense to create a new controller specifically for all actions about manipulating country data. Name this new controller as `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()` to build a DB query and retrieve all 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 for 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 most 5 rows in a page).
- Being 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, the `index` action renders a view named `index` and passes the country data as well as the pagination information to it.

2.5.5 Creating a View

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

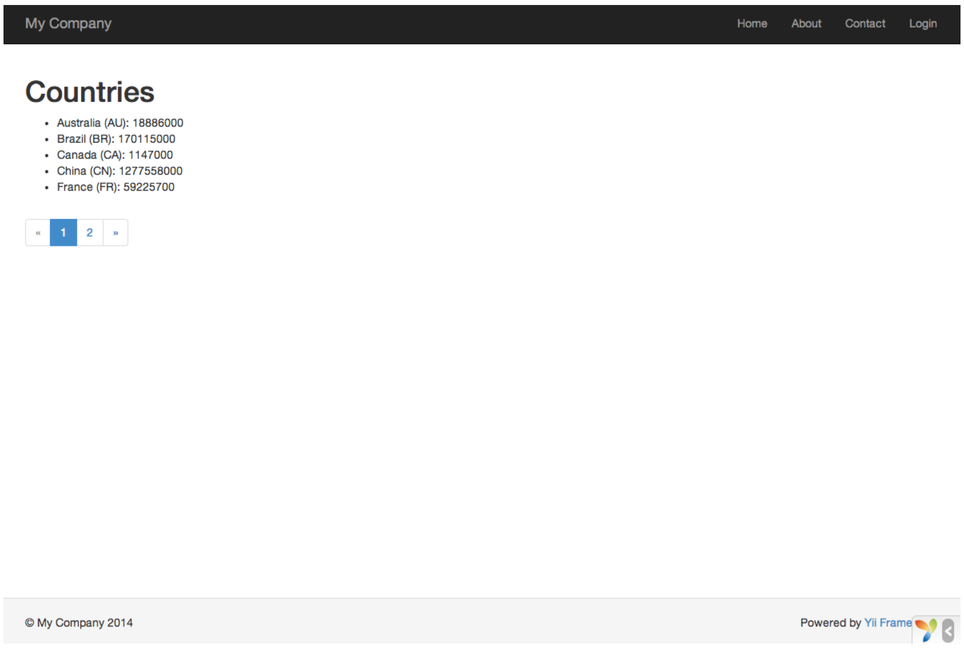
```
<?php
use yii\helpers\Html;
use yii\widgets\LinkPager;
?>
<h1>Countries</h1>
<ul>
<?php foreach ($countries as $country): ?>
    <li>
        <?= Html::encode("{$country->name} ({$country->code})") ?>:
        <?= $country->population ?>
    </li>
<?php endforeach; ?>
</ul>

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

The view consists of two parts. In the first part, the 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.

2.5.6 Trying it Out

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

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



Countries

- Australia (AU): 18886000
- Brazil (BR): 170115000
- Canada (CA): 1147000
- China (CN): 1277558000
- France (FR): 59225700

You will see a page showing five countries. And below the countries, you will see a pager with four buttons. If you click on the button "2", you will see that the page displays another five countries in the database. Observe more carefully and you will find the URL in the browser changes to

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

Behind the scene, `yii\data\Pagination` is playing the magic.

- Initially, `yii\data\Pagination` represents the first page, which sets the country query with the clause `LIMIT 5 OFFSET 0`. As a result, the first five countries will be fetched and displayed.
- The `yii\widgets\LinkPager` widget renders the page buttons using the URLs created by `yii\data\Pagination::createUrl()`. The URLs will contain the query parameter `page` representing 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` reads the `page` query parameter and sets the current page number 2. The new country query will thus have the clause `LIMIT 5 OFFSET 5` and return back the next five countries for display.

2.5.7 Summary

In this section, you have learned how to work with a database. You have 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 about the data in a DB table. As a matter of fact, the code you have just written can all be automatically generated using this tool.

2.6 Generating Code with Gii

In this section, we will describe how to use Gii to automatically generate the code that implements some common features. To achieve this goal, all you need is just to enter the needed information according to the instructions showing on the Gii Web pages.

Through this tutorial, you will learn

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

2.6.1 Starting Gii

Gii is provided by Yii in terms of a module. You can enable Gii by configuring it in the `yii\base\Application::modules` property of the application. In particular, you may find the following code is already given in the `config/web.php` file - the application configuration,

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

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

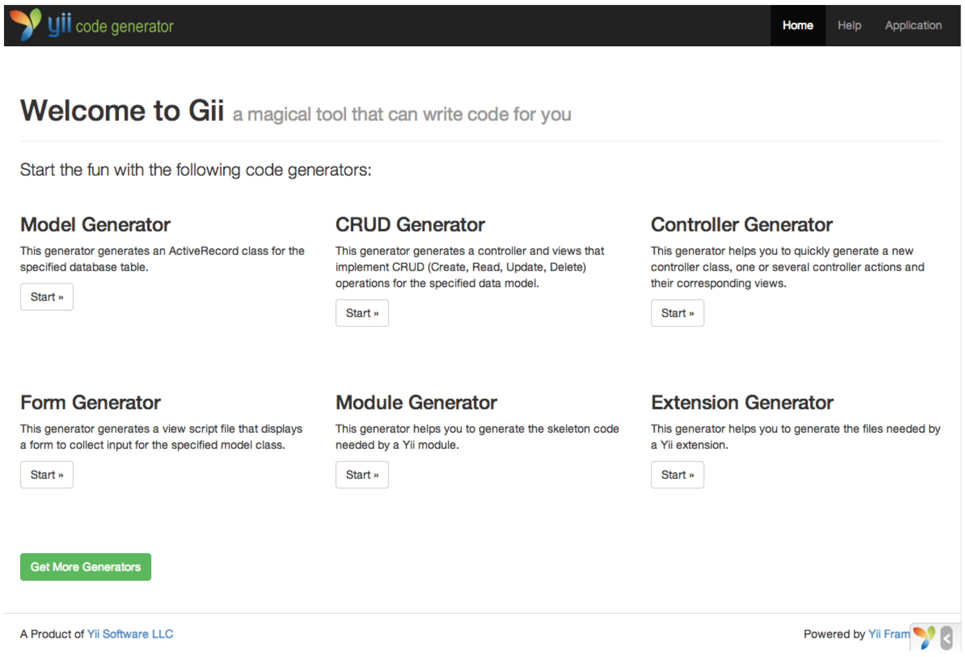
The above configuration states that when in development environment, the application should include a module named `gii` which is of class `yii\gii\Module`.

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');
```

Therefore, your application has already enabled Gii, and you can access it via the following URL:

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



Welcome to Gii a magical tool that can write code for you

Start the fun with the following code generators:

Model Generator
This generator generates an ActiveRecord class for the specified database table.
Start »

CRUD Generator
This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.
Start »

Controller Generator
This generator helps you to quickly generate a new controller class, one or several controller actions and their corresponding views.
Start »

Form Generator
This generator generates a view script file that displays a form to collect input for the specified model class.
Start »

Module Generator
This generator helps you to generate the skeleton code needed by a Yii module.
Start »

Extension Generator
This generator helps you to generate the files needed by a Yii extension.
Start »

Get More Generators

A Product of Yii Software LLC

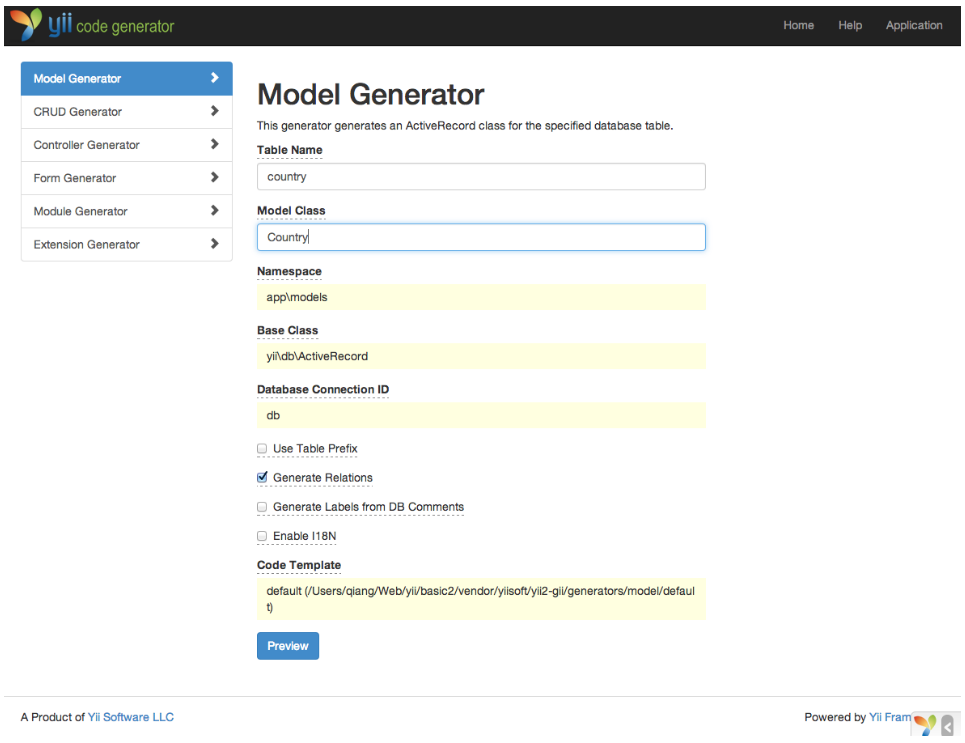
Powered by Yii Fram

2.6.2 Generating an Active Record Class

To use Gii to generate an Active Record class, select the "Model Generator" and fill out the form as follows,

- Table Name: `country`

- Model Class: `Country`



Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name: country

Model Class: Country

Namespace: app\models

Base Class: yii\db\ActiveRecord

Database Connection ID: db

☐ Use Table Prefix

☑ Generate Relations

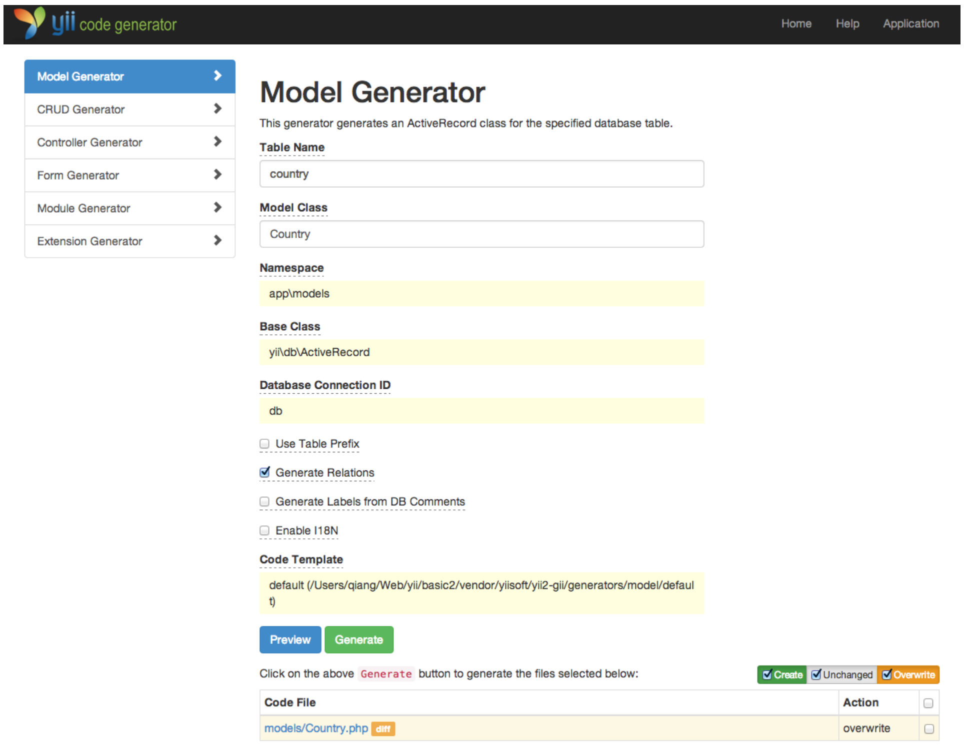
☐ Generate Labels from DB Comments

☐ Enable I18N

Code Template: default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default)

Click on the "Preview" button. You will see `models/Country.php` is listed in the result. You may click on it to preview its content.

Because in the last section, you have already created the same file `models/Country.php`, if you click the `diff` button next to the file name, you will see the difference between the code to be generated and the code that you have already written.


yii code generator

Home Help Application

- Model Generator
- CRUD Generator
- Controller Generator
- Form Generator
- Module Generator
- Extension Generator

Model Generator

This generator generates an ActiveRecord class for the specified database table.

Table Name: country

Model Class: Country

Namespace: app\models

Base Class: yii\db\ActiveRecord

Database Connection ID: db

- [ ] Use Table Prefix
- [✓] Generate Relations
- [ ] Generate Labels from DB Comments
- [ ] Enable I18N

Code Template: default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/model/default)

Preview Generate

Click on the above `Generate` button to generate the files selected below:

✓ Create ✓ Unchanged ✓ Overwrite

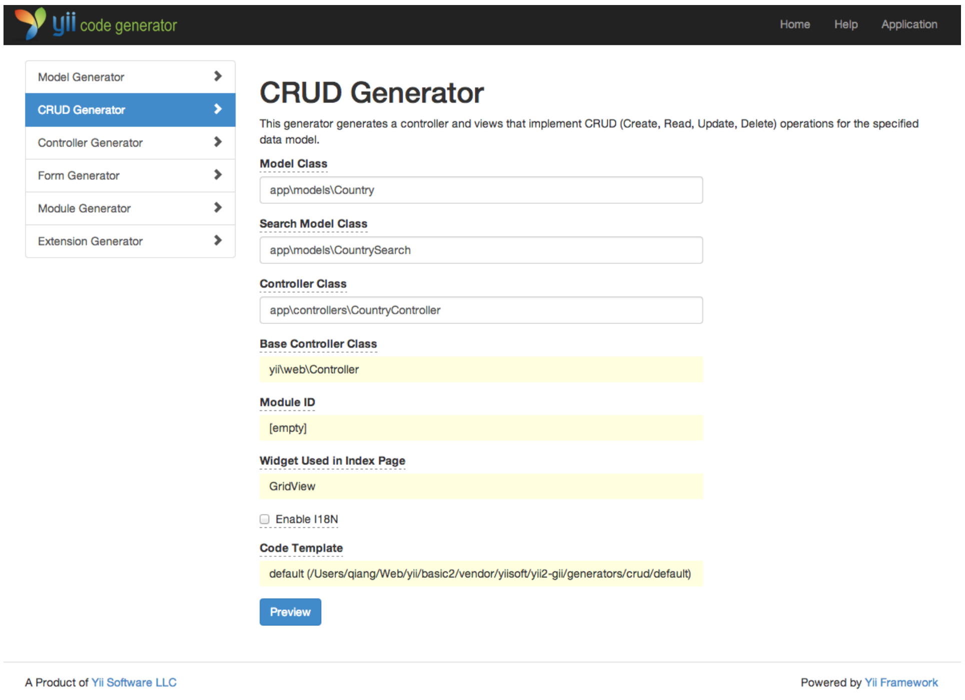
Code File	Action	
models/Country.php diff	overwrite	☐

Check the checkbox next to “overwrite” and then click on the “Generate” button. You will see a confirmation page indicating the code has been successfully generated and your existing `models/Country.php` is overwritten with the newly generated code.

2.6.3 Generating CRUD Code

To create CRUD code, select the “CRUD Generator”. Fill out the form as follows:

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



CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.

Model Class: app\models\Country

Search Model Class: app\models\CountrySearch

Controller Class: app\controllers\CountryController

Base Controller Class: yii\web\Controller

Module ID: [empty]

Widget Used in Index Page: GridView

☐ Enable I18N

Code Template: default (/Users/qiang/Web/yii/basic2/vendor/yiisoft/yii2-gii/generators/crud/default)

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

Make sure you have checked the overwrite checkboxes for both `controllers/CountryController.php` and `views/country/index.php` files. This is needed because you have already created these files in the previous section and you want to have them overwritten to have full CRUD support.

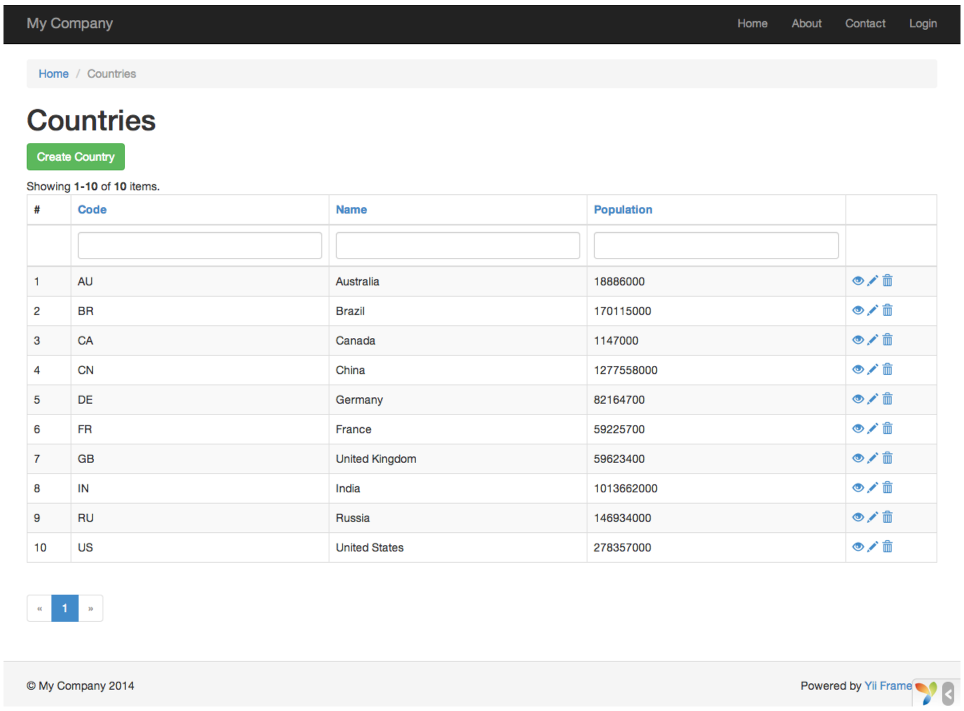
2.6.4 Trying it Out

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

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

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

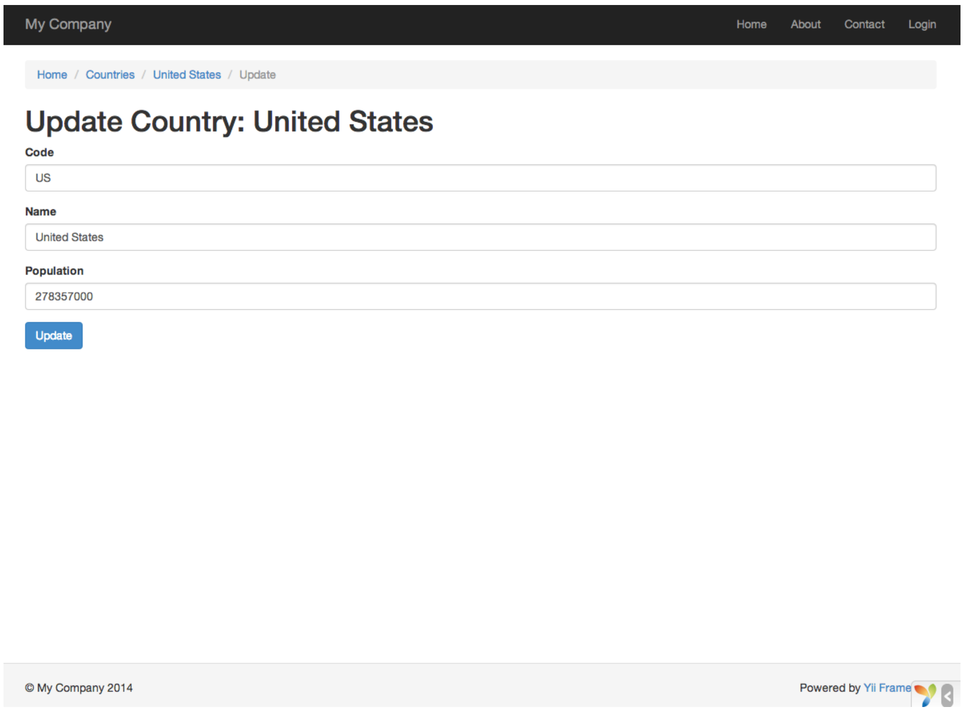
For each country displayed in the grid, you may choose to view its detail, update it or delete it. You may also click on the "Create Country" button on top of the grid to create a new country.



Countries

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	



Update Country: United States

Code	US
Name	United States
Population	278357000

The following is the list of the generated files in case you want to dig out how these features are implemented, or if you want 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.

2.6.5 Summary

In this section, you have learned how to use Gii to generate the code that implements a complete set of CRUD features regarding a database table.

2.7 Looking Ahead

To this end, you have created a complete Yii application, and you have learned how to implement some commonly needed features, such as getting data from users using an HTML form, fetching data from database and displaying it in a paginated fashion. You have also learned how to use Gii to generate code automatically, which turns programming into a task as simple as just filling out some forms. In this section, we will summarize the resources about Yii that help you be more productive when using Yii.

- Documentation
  - The Definitive Guide: As the name indicates, the guide precisely defines how Yii should work and gives you a general guidance about using Yii. It is the single most important Yii tutorial that you should read through 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.
  - The Wiki Articles: The wiki articles are written by Yii users based on their own experiences. Most of them are written like cookbook recipes which show how to solve particular problems using Yii. While the quality of these articles may be as good as the Definitive Guide, they are useful in that they cover broader topics and can often provide to you ready-to-use solutions.
  - Books

- Extensions[15]: Yii boasts a library of thousands of user-contributed extensions that can be easily plugged into your applications and make your application development even faster and easier.
- Community
  - Forum[16]
  - GitHub[17]
  - Facebook[18]
  - Twitter[19]
  - LinkedIn[20]

[15]http://www.yiiframework.com/extensions/
[16]http://www.yiiframework.com/forum/
[17]https://github.com/yiisoft/yii2
[18]https://www.facebook.com/groups/yiitalk/
[19]https://twitter.com/yiiframework
[20]https://www.linkedin.com/groups/yii-framework-1483367

Chapter 3

Application Structure

3.1 Overview

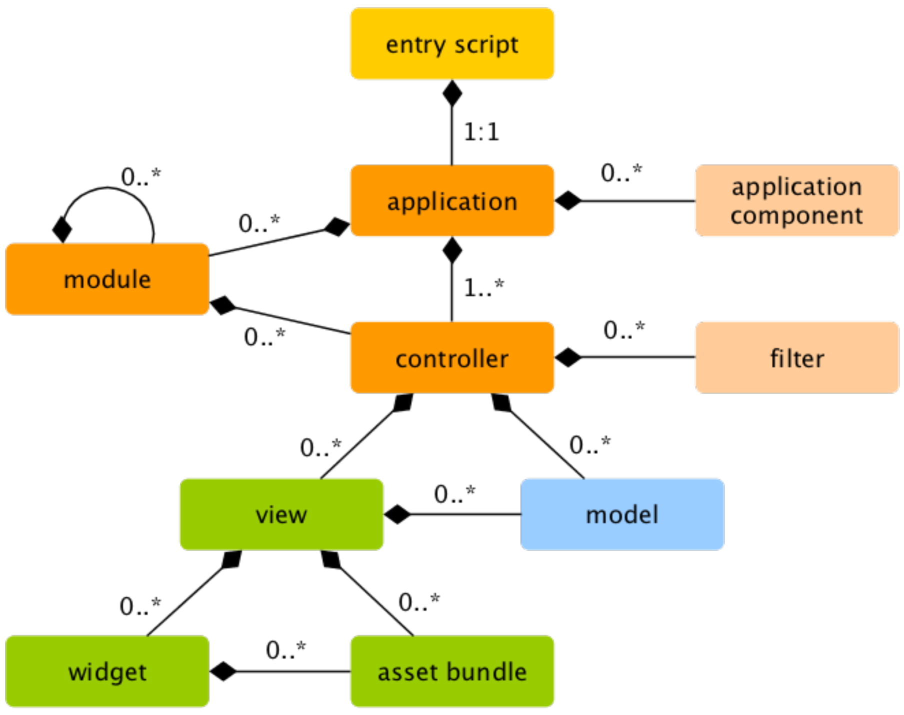
Yii applications are organized according to the model-view-controller (MVC)[1] design pattern. Models represent data, business logic and rules; views are output representation of models; and controllers take input and convert it to commands for models and views.

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:

[1] `http://wikipedia.org/wiki/Model-view-controller`



3.2 Entry Scripts

Entry scripts are the first chain 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[2];
- Include the `Yii` class file;

[2]`http://getcomposer.org/doc/01-basic-usage.md#autoloading`

- Load application configuration;
- Create and configure an application instance;
- Call `yii\base\Application::run()` to process the incoming request.

3.2.1 Web Applications

The following is the code in the entry script for the Basic Web Application 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();
```

3.2.2 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);

// fcgi doesn't have STDIN and STDOUT defined by default
defined('STDIN') or define('STDIN', fopen('php://stdin', 'r'));
defined('STDOUT') or define('STDOUT', fopen('php://stdout', 'w'));

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

// load application configuration
```

```
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);
```

3.2.3 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 has been 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.

3.3 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` and `yii\console\Application`. As the names indicate, the former mainly handles Web requests while the latter console command requests.

3.3.1 Application Configurations

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

```
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.

3.3.2 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` and `yii\base\Application::basePath`.

`yii\base\Application::id` The `yii\base\Application::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 alphanumeric characters only when specifying an application ID.

`yii\base\Application::basePath` The `yii\base\Application::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`, `controllers`, which contain source code corresponding to the MVC pattern.

You may configure the `yii\base\Application::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` 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` 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 such that you can define aliases in terms of application configurations instead of the method calls `Yii::setAlias()`.

`yii\base\Application::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()`. 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.

For example,

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

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

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

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

Another practical example is in the application configuration for the Basic Application Template, where the `debug` and `gii` modules are configured as bootstrap components when the application is running in 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 bootstrap components judiciously.

`yii\web\Application::catchAll` This property is supported by `yii\web\Application` 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',
    ],
]
```

`yii\base\Application::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 application 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` 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` 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 "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` 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 should the date be formatted.

It is recommended that you specify a language in terms of an IETF language tag[3]. 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` 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',
```

[3]http://en.wikipedia.org/wiki/IETF_language_tag

```
            'db' => 'db',
        ],
    ],
]
```

Please refer to the Modules section for more details.

`yii\base\Application::name` This property specifies the application name that may be displayed to end users. Unlike the `yii\base\Application::id` property which should take a unique value, the value of this property is mainly for display purpose and 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` 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 the 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 without touching any dependent code.

`yii\base\Application::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[4]. 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.

[4]`http://en.wikipedia.org/wiki/IETF_language_tag`

`yii\base\Application::timeZone` This property is provided as an alternative way of setting the default time zone of PHP runtime. By configuring this property, you are essentially calling the PHP function date_default_timezone_set()[5]. For example,

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

`yii\base\Application::version` This property specifies the version of the application. It defaults to `'1.0'`. You do not always 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 stipulate common conventions. However, you may still configure them in case you want to break the conventions.

`yii\base\Application::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 some legacy systems that use a lot of non-unicode data.

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

[5]`http://php.net/manual/en/function.date-default-timezone-set.php`

Composer[6] 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 like the following:

```
[
    'extensions' => [
        [
            'name' => 'extension name',
            'version' => 'version number',
            'bootstrap' => 'BootstrapClassName',  // optional, may also be a
     configuration 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 property, a `bootstrap` element may be specified with a bootstrap class name or a configuration array. An extension may also define a few aliases.

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

[6]`http://getcomposer.org`

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

`yii\base\Application::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` This property specifies the vendor directory managed by Composer[7]. 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 accessing to this path, Yii has predefined a path alias named `@vendor` for it.

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

3.3.3 Application Events

An application triggers several events during the lifecycle of handling an request. You may attach event handlers to these events in application configurations like the following,

```
[
    '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 process after the application instance is created. For example,

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

[7]http://getcomposer.org

```
    // ...
});
```

yii\base\Application::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_BEFORE_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` 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_REQUEST

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)(structure-controllers.md). 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 the following events will NOT be triggered.

`yii\base\Application::EVENT_BEFORE_REQUEST`

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](structure-controllers.md). 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.

3.3.4 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()` is called, which configures some high priority application properties, such as `yii\base\Application::basePath`.
   - Register the `yii\base\Application::errorHandler`.
   - Configure application properties.
   - `yii\base\Application::init()` is called which further calls `yii\base\Application::bootstrap()` to run bootstrap components.
3. The entry script calls `yii\base\Application::run()` to run the application:
   - Trigger the `yii\base\Application::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.

    - Send response to the end user.

4. The entry script receives the exit status from the application and completes the request processing.

3.4 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::$app->db` to get the `yii\db\Connection`, and `\Yii::$app->cache` to get the `yii\caching\Cache` registered with the application.

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.

3.4.1 Core Application Components

Yii defines a set of *core* application components with fixed IDs and default configurations. For example, the `yii\web\Application::request` component is used to collect information about a user request and resolve it into a route; the `yii\base\Application::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`: manages asset bundles and asset publishing. Please refer to the Managing Assets section for more details.

- `yii\db\Connection`: 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`: handles PHP errors and exceptions. Please refer to the Handling Errors section for more details.

- `yii\base\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`: supports message translation and formatting. Please refer to the Internationalization section for more details.

- `yii\log\Dispatcher`: manages log targets. Please refer to the Logging section for more details.

- `yii\swiftmailer\Mailer`: supports mail composing and sending. Please refer to the Mailing section for more details.

- `yii\base\Application::response`: represents the response being sent to end users. Please refer to the Responses section for more details.

- `yii\base\Application::request`: represents the request received from end users. Please refer to the Requests section for more details.
- `yii\web\Session`: represents the session information. This component is only available in `yii\web\Application`. Please refer to the Sessions and Cookies section for more details.
- `yii\web\UrlManager`: supports URL parsing and creation. Please refer to the URL Parsing and Generation section for more details.
- `yii\web\User`: represents the user authentication information. This component is only available in `yii\web\Application` Please refer to the Authentication section for more details.
- `yii\web\View`: supports view rendering. Please refer to the Views section for more details.

3.5 Controllers

Controllers are part of the MVC[8] architecture. They are objects 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.

Controllers are composed by *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`:

```
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;
        }
```

[8] http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

```
        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 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.

3.5.1 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 t hat 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 how routes are resolved into actions, please refer to the Routing section.

3.5.2 Creating Controllers

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

```
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, dashes and forward slashes. For example, `article`, `post-comment`, `admin/post2-comment` are all valid controller IDs, while `article?`, `PostComment`, `admin\post` are not.

The dashes in a controller ID are used to separate words, while the forward slashes to organize controllers in sub-directories.

Controller Class Naming

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

- Turn the first letter in each word separated by dashes 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.
- Remove dashes and replace any forward slashes with backward slashes.
- Append the suffix `Controller`.
- And prepend the `yii\base\Application::controllerNamespace`.

The followings are some examples, assuming the `yii\base\Application::controllerNamespace` takes the default value `app\controllers`:

- `article` derives `app\controllers\ArticleController`;
- `post-comment` derives `app\controllers\PostCommentController`;
- `admin/post2-comment` derives `app\controllers\admin\Post2CommentController`.

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/post2-comment` controller should be in `@app/controllers/admin/Post2CommentController.php`.

> Info: The last example `admin/post2-comment` shows how you can put a controller under a sub-directory of the `yii\base\Application::controllerNamespace`. 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 `yii\base\Application::controllerMap` to overcome the constraints of the controller IDs and class names described above. This is mainly useful when you are using some third-party controllers which you do not control over their class names.

You may configure `yii\base\Application::controllerMap` in the application configuration like the following:

```
[
    '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`, its value is `'site'`, while for `yii\console\Application`, it is `help`. Therefore, if a URL is `http://hostname/index.php`, it means the `site` controller will handle the request.

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

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

3.5.3 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 about 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 dashes. The dashes in an actionID are used to separate words. For example, `view`, `update2`, `comment-post` are all valid action IDs, while `view?`, `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 class.

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 criteria:

- Turn the first letter in each word of the action ID into upper case;
- Remove dashes;
- 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 its 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
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 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 as the response to end users.

- For `yii\web\Application`, 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` 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()` method returns a response object):

```
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`, the value of each action parameter is retrieved from `$_GET` using the parameter name as the key; for `yii\console\Application`, 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');
    }
}
```

3.5.4 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` will be used.
   - If the action ID is found in the `yii\base\Controller::actions()`, 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()` 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.

3.5.5 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.

In summary, 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 models;
- should avoid embedding HTML or other presentational code - this is better done in views.

3.6 Models

Models are part of the MVC[9] 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.

3.6.1 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:

[9] `http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller`

```
$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[10] and ArrayIterator[11] by `yii\base\Model`:

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

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

// iterate attributes
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.

[10]http://php.net/manual/en/class.arrayaccess.php
[11]http://php.net/manual/en/class.arrayiterator.php

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 "Label"
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()` 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.

3.6.2 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 = 'login';

// scenario is set through configuration
$model = new 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
{
    public function scenarios()
    {
        return [
            'login' => ['username', 'password'],
            '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:

```
namespace app\models;

use yii\db\ActiveRecord;

class User extends ActiveRecord
{
    public function scenarios()
    {
        $scenarios = parent::scenarios();
        $scenarios['login'] = ['username', 'password'];
        $scenarios['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.

3.6.3 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' => 'register'],

        // username and password are required in "login" scenario
        [['username', 'password'], 'required', 'on' => '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`.

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()`.

3.6.4 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` 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 [
        'login' => ['username', 'password'],
        '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 [
        '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;
```

3.6.5 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. In the first step, models are converted into arrays; in the second step, 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()` and/or `yii\base\Model::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 backward 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`.

3.6.6 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 mode 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 crete 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 Application 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.

3.7 Views

Views are part of the MVC[12] architecture. They are responsible for presenting data to end users. In a Yii application, the view layer is composed by view templates and view components. The former contains presentational code (e.g. HTML), while the latter provides common view-related features and is responsible for turning view templates into response content. We often use "views" to refer to view templates.

3.7.1 View Templates

You turn a view template into response content by pushing model data into the template and rendering it. For example, in the `post/view` action below, the `view` template is rendered with the `$model` data. The rendering result is a string which is returned by the action as the response content.

```
namespace app\controllers;
```

[12]http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller

```
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,
        ]);
    }
}
```

Rendering Views in Controllers

Rendering Views in Widgets

Rendering Views in Views

View Names

By default, a view template is simply a PHP script file (called *view file*). Like models and controllers, view files are usually organized under the so-called *view paths*. For views rendered by controllers that directly belong to an application, they are located under the `@app/views` path.

- applications:
- modules:
- controllers:
- widgets and other components:

When rendering a view template, you can specify it using a view name, a view file path, or an alias to the view file path.

If a view name is used, it will be resolved into a view file path according to the current `yii\base\View::context` using one of the following rules:

- If the view name starts with double slashes `//`, the corresponding view file path is considered to be `@app/views/ViewName`. For example, `//site/about` will be resolved into `@app/views/site/about`.

- 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` 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`, if the currently active module is `user`. If there is no active module, the view file path would be `@app/views/user/create`.
- If the view is rendered with a `yii\base\View::context`, the view file path will be prefixed with the `yii\base\ViewContextInterface::getViewPath()`. For example, `site/about` will be resolved into `@app/views/site/about` if the context is the `SiteController`.
- If a view was previously rendered, the directory containing that view file will be prefixed to the new view name.

3.7.2 Layouts

Nested Layouts

Accessing Data

3.7.3 View Components

3.7.4 Creating Views

Setting page title

Adding meta tags

Registering link tags

Registering CSS

Registering scripts

Static Pages

Assets

3.7.5 Alternative Template Engines

3.7.6 Basics

By default, Yii uses PHP in view templates to generate content and elements. A web application view typically contains some combination of HTML, along with PHP `echo`, `foreach`, `if`, and other basic constructs. Using complex PHP code in views is considered to be bad practice. When complex logic and functionality is needed, such code should either be moved to a controller or a widget.

The view is typically called from controller action using the `yii\base\Controller::render()` method:

```
public function actionIndex()
{
    return $this->render('index', ['username' => 'samdark']);
}
```

The first argument to `yii\base\Controller::render()` is the name of the view to display. In the context of the controller, Yii will search for its views in `views/site/` where `site` is the controller ID. For details on how the view name is resolved, refer to the `yii\base\Controller::render()` method.

The second argument to `yii\base\Controller::render()` is a data array of key-value pairs. Through this array, data can be passed to the view, making the value available in the view as a variable named the same as the corresponding key.

The view for the action above would be `views/site/index.php` and can be something like:

```
<p>Hello, <?= $username ?>!</p>
```

Any data type can be passed to the view, including arrays or objects.

Besides the above `yii\web\Controller::render()` method, the `yii\web\Controller` class also provides several other rendering methods. Below is a summary of these methods:

- `yii\web\Controller::render()`: renders a view and applies the layout to the rendering result. This is most commonly used to render a complete page.

- `yii\web\Controller::renderPartial()`: renders a view without applying any layout. This is often used to render a fragment of a page.

- `yii\web\Controller::renderAjax()`: renders a view without applying any layout, and injects all registered JS/CSS scripts and files. It is most commonly used to render an HTML output to respond to an AJAX request.

- `yii\web\Controller::renderFile()`: renders a view file. This is similar to `yii\web\Controller::renderPartial()` except that it takes the file path of the view instead of the view name.

3.7.7 Widgets

Widgets are self-contained building blocks for your views, a way to combine complex logic, display, and functionality into a single component. A widget:

- May contain advanced PHP programming

- Is typically configurable
- Is often provided data to be displayed
- Returns HTML to be shown within the context of the view

There are a good number of widgets bundled with Yii, such as active form, breadcrumbs, menu, and wrappers around bootstrap component framework. Additionally there are extensions that provide more widgets, such as the official widget for jQueryUI[13] components.

In order to use a widget, your view file would do the following:

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

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

In the first example in the code above, the `yii\base\Widget::widget()` method is used to invoke a widget that just outputs content. In the second example, `yii\base\Widget::begin()` and `yii\base\Widget::end()` are used for a widget that wraps content between method calls with its own output. In case of the form this output is the `<form>` tag with some properties set.

3.7.8 Security

One of the main security principles is to always escape output. If violated it leads to script execution and, most probably, to cross-site scripting known as XSS leading to leaking of admin passwords, making a user to automatically perform actions etc.

Yii provides a good tool set in order to help you escape your output. The very basic thing to escape is a text without any markup. You can deal with it like the following:

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

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

[13] `http://www.jqueryui.com`

When you want to render HTML it becomes complex so we're delegating the task to excellent HTMLPurifier[14] library which is wrapped in Yii as a helper `yii\helpers\HtmlPurifier`:

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

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

Note that besides HTMLPurifier does excellent job making output safe it's not very fast so consider caching result.

3.7.9 Alternative template languages

There are official extensions for Smarty[15] and Twig[16]. In order to learn more refer to Using template engines section of the guide.

3.7.10 Using View object in templates

An instance of `yii\web\View` component is available in view templates as `$this` variable. Using it in templates you can do many useful things including setting page title and meta, registering scripts and accessing the context.

Setting page title

A common place to set page title are view templates. Since we can access view object with `$this`, setting a title becomes as easy as:

```
$this->title = 'My page title';
```

Adding meta tags

Adding meta tags such as encoding, description, keywords is easy with view object as well:

```
$this->registerMetaTag(['encoding' => 'utf-8']);
```

The first argument is an map of `<meta>` tag option names and values. The code above will produce:

```
<meta encoding="utf-8">
```

Sometimes there's a need to have only a single tag of a type. In this case you need to specify the second argument:

[14]`http://htmlpurifier.org/`
[15]`http://www.smarty.net/`
[16]`http://twig.sensiolabs.org/`

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

If there are multiple calls with the same value of the second argument (`meta-description` in this case), the latter will override the former and only a single tag will be rendered:

```
<meta name="description" content="This website is about funny raccoons.">
```

Registering link tags

`<link>` tag is useful in many cases such as customizing favicon, pointing to RSS feed or delegating OpenID to another server. Yii view object has a method to work with these:

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

The code above will result in

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

Same as with meta tags you can specify additional argument to make sure there's only one link of a type registered.

Registering CSS

You can register CSS using `yii\web\View::registerCss()` or `yii\web\View::registerCssFile()`. The former registers a block of CSS code while the latter registers an external CSS file. For example,

```
$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 third argument. If you need to make sure there's only a single style tag use fourth argument as was mentioned in meta tags description.

```
$this->registerCssFile("http://example.com/css/themes/black-and-white.css",
    [BootstrapAsset::className()], ['media' => 'print'], 'css-print-theme')
    ;
```

The code above will add a link to CSS file to the head section of the page.

- The first argument specifies the CSS file to be registered.
- The second argument specifies that this CSS file depends on `yii\bootstrap\BootstrapAsset`, meaning it will be added AFTER the CSS files in `yii\bootstrap\BootstrapAsset`. Without this dependency specification, the relative order between this CSS file and the `yii\bootstrap\BootstrapAsset` CSS files would be undefined.
- The third argument specifies the attributes for the resulting `<link>` tag.
- 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 that you use asset bundles to register external CSS files rather than using `yii\web\View::registerCssFile()`. Using asset bundles allows you to combine and compress multiple CSS files, which is desirable for high traffic websites.

Registering scripts

With the `yii\web\View` object you can register scripts. There are two dedicated methods for it: `yii\web\View::registerJs()` for inline scripts and `yii\web\View::registerJsFile()` for external scripts. Inline scripts are useful for configuration and dynamically generated code. The method for adding these can be used as follows:

```
$this->registerJs("var options = ".json_encode($options).";", View::POS_END,
    'my-options');
```

The first argument is the actual JS code we want to insert into the page. The second argument determines where script should be inserted into the page. Possible values are:

- `yii\web\View::POS_HEAD` for head section.
- `yii\web\View::POS_BEGIN` for right after opening `<body>`.
- `yii\web\View::POS_END` for right before closing `</body>`.
- `yii\web\View::POS_READY` for executing code on document `ready` event. This will register `yii\web\JqueryAsset` automatically.
- `yii\web\View::POS_LOAD` for executing code on document `load` event. This will register `yii\web\JqueryAsset` automatically.

The last argument is a unique script ID that is used to identify code block and replace 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.

An external script can be added like the following:

```
$this->registerJsFile('http://example.com/js/main.js', [JqueryAsset::
    className()]);
```

The arguments for `yii\web\View::registerJsFile()` are similar to those for `yii\web\View::registerCssFile()`. In the above example, we register the `main.js` file with the dependency on `JqueryAsset`. This means the `main.js` file will be added AFTER `jquery.js`. Without this dependency specification, the relative order between `main.js` and `jquery.js` would be undefined.

Like for `yii\web\View::registerCssFile()`, it is also highly recommended that you use asset bundles to register external JS files rather than using `yii\web\View::registerJsFile()`.

Registering asset bundles

As was mentioned earlier it's preferred to use asset bundles instead of using CSS and JavaScript directly. You can get details on how to define asset bundles in asset manager section of the guide. As for using already defined asset bundle, it's very straightforward:

```
\frontend\assets\AppAsset::register($this);
```

Layout

A layout is a very convenient way to represent the part of the page that is common for all or at least for most pages generated by your application. Typically it includes `<head>` section, footer, main menu and alike elements. You can find a fine example of the layout in a basic application template. Here we'll review the very basic one without any widgets or extra markup.

```
<?php
use yii\helpers\Html;
?>
<?php $this->beginPage() ?>
<!DOCTYPE html>
<html lang="<?= Yii::$app->language ?>">
<head>
    <meta charset="<?= Yii::$app->charset ?>"/>
    <title><?= Html::encode($this->title) ?></title>
    <?php $this->head() ?>
</head>
<body>
<?php $this->beginBody() ?>
    <div class="container">
        <?= $content ?>
    </div>
    <footer class="footer">&copy; 2013 me :)</footer>
<?php $this->endBody() ?>
</body>
</html>
<?php $this->endPage() ?>
```

In the markup above there's some code. First of all, `$content` is a variable that will contain result of views rendered with controller's `$this->render()` method.

We are importing `yii\helpers\Html` helper via standard PHP `use` statement. This helper is typically used for almost all views where one need to escape outputted data.

Several special methods such as `yii\web\View::beginPage()`/`yii\web\View::endPage()`, `yii\web\View::head()`, `yii\web\View::beginBody()`/`yii\web\View::endBody()` are triggering page rendering events that are used for registering scripts, links and process page in many other ways. Always include these in your layout in order for the rendering to work correctly.

By default layout is loaded from `views/layouts/main.php`. You may change it at controller or module level by setting different value to `layout` property.

In order to pass data from controller to layout, that you may need for breadcrumbs or similar elements, use view component params property. In controller it can be set as:

```
$this->view->params['breadcrumbs'][] = 'Contact';
```

In a view it will be:

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

In layout file the value can be used like the following:

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

You may also wrap the view render result into a layout using `yii\base\View::beginContent()`, `yii\base\View::endContent()`. This approach can be used while applying nested layouts:

```
<?php $this->beginContent('//layouts/overall') ?>
<div class="content">
    <?= $content ?>
<div>
<?php $this->endContent() ?>
```

Partials

Often you need to reuse some HTML markup in many views and often it's too simple to create a full-featured widget for it. In this case you may use partials.

Partial is a view as well. It resides in one of directories under `views` and by convention is often started with `_`. For example, we need to render a list of user profiles and, at the same time, display individual profile elsewhere.

First we need to define a partial for user profile in `_profile.php`:

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

<div class="profile">
    <h2><?= Html::encode($username) ?></h2>
    <p><?= Html::encode($tagline) ?></p>
</div>
```

Then we're using it in `index.php` view where we display a list of users:

```
<div class="user-index">
    <?php
    foreach ($users as $user) {
        echo $this->render('_profile', [
            'username' => $user->name,
            'tagline' => $user->tagline,
        ]);
    }
    ?>
</div>
```

Same way we can reuse it in another view displaying a single user profile:

```
echo $this->render('_profile', [
    'username' => $user->name,
    'tagline' => $user->tagline,
]);
```

When you call `render()` to render a partial in a current view, you may use different formats to refer to the partial. The most commonly used format is the so-called relative view name which is as shown in the above example. The partial view file is relative to the directory containing the current view. If the partial is located under a subdirectory, you should include the subdirectory name in the view name, e.g., `public/_profile`.

You may use path alias to specify a view, too. For example, `@app/views/common/_profile`.

And you may also use the so-called absolute view names, e.g., `/user/_profile`, `//user/_profile`. An absolute view name starts with a single slashes or double slashes. If it starts with a single slash, the view file will be looked for under the view path of the currently active module. Otherwise, it will will be looked for under the application view path.

Accessing context

Views are generally used either by controller or by widget. In both cases the object that called view rendering is available in the view as `$this->context`.

For example if we need to print out the current internal request route in a view rendered by controller we can use the following:

```
echo $this->context->getRoute();
```

Static Pages

If you need to render static pages you can use class `ViewAction`. It represents an action that displays a view according to a user-specified parameter.

Usage of the class is simple. In your controller use the class via `actions` method:

```
class SiteController extends Controller
{
    public function actions()
    {
        return [
            'static' => [
                'class' => '\yii\web\ViewAction',
            ],
        ];
    }

    //...
}
```

Then create `index.php` in `@app/views/site/pages/`:

```
//index.php
<h1>Hello, I am a static page!</h1>
```

That's it. Now you can try it using `/index.php?r=site/static`.

By default, the view being displayed is specified via the `view` GET parameter. If you open `/index.php?r=site/static?&view=about` then `@app/views/site/pages/about.php` view file will be used.

If not changed or specified via GET defaults are the following:

- GET parameter name: `view`.
- View file used if parameter is missing: `index.php`.
- Directory where views are stored (`viewPrefix`): `pages`.
- Layout for the page rendered matches the one used in controller.

For more information see `yii\web\ViewAction`.

Caching blocks

To learn about caching of view fragments please refer to caching section of the guide.

3.7.11 Customizing View component

Since view is also an application component named `view` you can replace it with your own component that extends from `yii\base\View` or `yii\web\View`. It can be done via application configuration file such as `config/web.php`:

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

You may apply some action filters to controller actions to accomplish tasks such as determining who can access the current action, decorating the result of the action, etc.

An action filter is an instance of a class extending `yii\base\ActionFilter`.

To use an action filter, attach it as a behavior to a controller or a module. The following example shows how to enable HTTP caching for the `index` action:

```
public function behaviors()
{
    return [
        'httpCache' => [
            'class' => \yii\filters\HttpCache::className(),
            'only' => ['index'],
            'lastModified' => function ($action, $params) {
                $q = new \yii\db\Query();
                return $q->from('user')->max('updated_at');
            },
        ],
    ];
}
```

You may use multiple action filters at the same time. These filters will be applied in the order they are declared in `behaviors()`. If any of the filter cancels the action execution, the filters after it will be skipped.

When you attach a filter to a controller, it can be applied to all actions of that controller; If you attach a filter to a module (or application), it can be applied to the actions of any controller within that module (or application).

To create a new action filter, extend from `yii\base\ActionFilter` and override the `yii\base\ActionFilter::beforeAction()` and `yii\base\ActionFilter::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()` determines whether an action should be executed or not. If `beforeAction()` of a filter returns false, the filters after this one will be skipped and the action will not be executed.

The authorization section of this guide shows how to use the `yii\filters\AccessControl` filter, and the caching section gives more details about the `yii\filters\PageCache` and `yii\filters\HttpCache` filters. These built-in filters are also good references when you learn to create your own filters.

Error: not existing file: structure-widgets.md

Error: not existing file: structure-modules.md

3.8 Managing assets

Note: This section is under development.

An asset in Yii is a file that is included into the page. It could be CSS, JavaScript or any other file. The framework provides many ways to work with assets from basics such as adding `<script src="...">` tags for a file which is covered by the View section, to advanced usage such as publishing files that are not under the webservers document root, resolving JavaScript dependencies or minifying CSS, which we will cover in the following.

3.8.1 Declaring asset bundles

In order to define a set of assets the belong together and should be used on the website you declare a class called an "asset bundle". The bundle defines a set of asset files and their dependencies on other asset bundles.

Asset files can be located under the webservers accessable directory but also hidden inside of application or vendor directories. If the latter, the asset bundle will care for publishing them to a directory accessible by the webserver so they can be included in the website. This feature is useful for extensions so that they can ship all content in one directory and make installation easier for you.

To define an asset you create a class extending from `yii\web\AssetBundle` and set the properties according to your needs. Here you can see an example asset definition which is part of the basic application template, the `AppAsset` asset bundle class. It defines assets required application wide:

```php
<?php

use yii\web\AssetBundle as 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',
    ];
}
```

In the above `$basePath` specifies web-accessible directory assets are served from. It is a base for relative `$css` and `$js` paths i.e. `@webroot/css/site.css` for `css/site.css`. Here `@webroot` is an alias that points to application's `web` directory.

`$baseUrl` is used to specify base URL for the same relative `$css` and `$js` i.e. `@web/css/site.css` where `@web` is an alias that corresponds to your website base URL such as `http://example.com/`.

In case you have asset files under a non web accessible directory, that is the case for any extension, you need to specify `$sourcePath` instead of `$basePath` and `$baseUrl`. **All files** from the source path will be copied or symlinked to the `web/assets` directory of your application prior to being registered. In this case `$basePath` and `$baseUrl` are generated automatically at the time of publishing the asset bundle. This is the way to work with assets when you want to publish the whole directory no matter what's in be it images, webfonts etc.

> **Note:** do not use the `web/assets` path to put your own files in it. It is meant to be used only for asset publishing. When you create files that are already in web accessable directory put them in folders like `web/css` or `web/js`.

Dependencies on other asset bundles are specified via `$depends` property. It is an array that contains fully qualified class names of bundle classes that should be published in order for this bundle to work properly. Javascript and CSS files for `AppAsset` are added to the header after the files of `yii\web\YiiAsset` and `yii\bootstrap\BootstrapAsset` in this example.

Here `yii\web\YiiAsset` adds Yii's JavaScript library while `yii\bootstrap\BootstrapAsset` includes Bootstrap[17] frontend framework.

Asset bundles are regular classes so if you need to define another one, just create alike class with unique name. This class can be placed anywhere but the convention for it is to be under `assets` directory of the application.

Additionally you may specify `$jsOptions`, `$cssOptions` and `$publishOptions` that will be passed to `yii\web\View::registerJsFile()`, `yii\web\View::registerCssFile()` and `yii\web\AssetManager::publish()` respectively during registering and publising an asset.

Language-specific asset bundle

If you need to define an asset bundle that includes JavaScript file depending on the language you can do it the following way:

```
class LanguageAsset extends AssetBundle
{
    public $language;
    public $sourcePath = '@app/assets/language';
    public $js = [
    ];
```

[17]`http://getbootstrap.com`

```
    public function registerAssetFiles($view)
    {
        $language = $this->language ? $this->language : Yii::$app->language;
        $this->js[] = 'language-' . $language . '.js';
        parent::registerAssetFiles($view);
    }
}
```

In order to set language use the following code when registering an asset bundle in a view:

```
LanguageAsset::register($this)->language = $language;
```

3.8.2 Registering asset bundle

Asset bundle classes are typically registered in view files or widgets that depend on the css or javascript files for providing its functionality. An exception to this is the `AppAsset` class defined above which is added in the applications main layout file to be registered on any page of the application. Registering an asset bundle is as simple as calling the `yii\web\AssetBundle::register()` method:

```
use app\assets\AppAsset;
AppAsset::register($this);
```

Since we're in a view context `$this` refers to `View` class. To register an asset inside of a widget, the view instance is available as `$this->view`:

```
AppAsset::register($this->view);
```

> Note: If there is a need to modify third party asset bundles it is recommended to create your own bundles depending on third party ones and use CSS and JavaScript features to modify behavior instead of editing files directly or copying them over.

3.8.3 Overriding asset bundles

Sometimes you need to override some asset bundles application wide. A good example is loading jQuery from CDN instead of your own server. In order to do it we need to configure `assetManager` application component via config file. In case of basic application it is `config/web.php`:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'bundles' => [
                'yii\web\JqueryAsset' => [
                     'sourcePath' => null,
                     'js' => ['//ajax.googleapis.com/ajax/libs/jquery/1.8.3/
    jquery.min.js']
                ],
            ],
        ],
    ],
];
```

In the above we're adding asset bundle definitions to the `yii\web\AssetManager::bundles` property of asset manager. Keys are fully qualified class names to asset bundle classes we want to override while values are key-value arrays of class properties and corresponding values to set.

Setting `sourcePath` to `null` tells asset manager not to copy anything while `js` overrides local files with a link to CDN.

3.8.4 Enabling symlinks

Asset manager is able to use symlinks instead of copying files. It is turned off by default since symlinks are often disabled on shared hosting. If your hosting environment supports symlinks you certainly should enable the feature via application config:

```
return [
    // ...
    'components' => [
        'assetManager' => [
            'linkAssets' => true,
        ],
    ],
];
```

There are two main benefits in enabling it. First it is faster since no copying is required and second is that assets will always be up to date with source files.

3.8.5 Compressing and combining assets

To improve application performance you can compress and then combine several CSS or JS files into lesser number of files therefore reducing number of HTTP requests and overall download size needed to load a web page. Yii provides a console command that allows you to do both.

Preparing configuration

In order to use `asset` command you should prepare a configuration first. A template for it can be generated using

```
yii asset/template /path/to/myapp/config.php
```

The template itself 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
    }',
    // The list of asset bundles to compress:
    'bundles' => [
        // 'yii\web\YiiAsset',
        // 'yii\web\JqueryAsset',
    ],
    // Asset bundle for compression output:
    'targets' => [
        'app\config\AllAsset' => [
            'basePath' => 'path/to/web',
            'baseUrl' => '',
            'js' => 'js/all-{hash}.js',
            'css' => 'css/all-{hash}.css',
        ],
    ],
    // Asset manager configuration:
    'assetManager' => [
        'basePath' => __DIR__,
        'baseUrl' => '',
    ],
];
```

In the above keys are `properties` of `AssetController`. `bundles` list contains bundles that should be compressed. These are typically what's used by application. `targets` contains a list of bundles that define how resulting files will be written. In our case we're writing everything to `path/to/web` that can be accessed like `http://example.com/` i.e. it is website root directory.

> Note: in the console environment some path aliases like '@webroot' and '@web' may not exist, so corresponding paths inside the configuration should be specified directly.

JavaScript files are combined, compressed and written to `js/all-{hash}.js` where {hash} is replaced with the hash of the resulting file.

`jsCompressor` and `cssCompressor` are console commands or PHP callbacks, which should perform JavaScript and CSS files compression correspondingly. You should adjust these values according to your environment. By default

Yii relies on Closure Compiler[18] for JavaScript file compression, and on YUI Compressor[19]. You should install this utilities manually, if you wish to use them.

Providing compression tools

The command relies on external compression tools that are not bundled with Yii so you need to provide CSS and JS compressors which are correspondingly specified via `cssCompressor` and `jsCompression` properties. If compressor is specified as a string it is treated as a shell command template which should contain two placeholders: `{from}` that is replaced by source file name and `{to}` that is replaced by output file name. Another way to specify compressor is to use any valid PHP callback.

By default for JavaScript compression Yii tries to use Google Closure compiler[20] that is expected to be in a file named `compiler.jar`.

For CSS compression Yii assumes that YUI Compressor[21] is looked up in a file named `yuicompressor.jar`.

In order to compress both JavaScript and CSS, you need to download both tools and place them under the directory containing your `yii` console bootstrap file. You also need to install JRE in order to run these tools.

You may customize the compression commands (e.g. changing the location of the jar files) in the `config.php` file like the following,

```
return [
       'cssCompressor' => 'java -jar path.to.file\yuicompressor.jar  --type
    css {from} -o {to}',
       'jsCompressor' => 'java -jar path.to.file\compiler.jar --js {from} --
    js_output_file {to}',
];
```

where `{from}` and `{to}` are tokens that will be replaced with the actual source and target file paths, respectively, when the `asset` command is compressing every file.

Performing compression

After configuration is adjusted you can run the `compress` action, using created config:

```
yii asset /path/to/myapp/config.php /path/to/myapp/config/assets_compressed.
    php
```

Now processing takes some time and finally finished. You need to adjust your web application config to use compressed assets file like the following:

[18]https://developers.google.com/closure/compiler/
[19]https://github.com/yui/yuicompressor/
[20]https://developers.google.com/closure/compiler/
[21]https://github.com/yui/yuicompressor/

```
'components' => [
    // ...
    'assetManager' => [
        'bundles' => require '/path/to/myapp/config/assets_compressed.php',
    ],
],
```

3.8.6 Using asset converter

Instead of using CSS and JavaScript directly often developers are using their improved versions such as LESS or SCSS for CSS or Microsoft TypeScript for JavaScript. Using these with Yii is easy.

First of all, corresponding compression tools should be installed and should be available from where `yii` console bootstrap file is. The following lists file extensions and their corresponding conversion tool names that Yii converter recognizes:

- LESS: `less - lessc`
- SCSS: `scss, sass - sass`
- Stylus: `styl - stylus`
- CoffeeScript: `coffee - coffee`
- TypeScript: `ts - tsc`

So if the corresponding tool is installed you can specify any of these in asset bundle:

```
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',
    ];
}
```

In order to adjust conversion tool call parameters or add new ones you can use application config:

```
// ...
'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've left two types of extra file extensions. First one is `less` that can be specified in `css` part of an asset bundle. Conversion is performed via running `lessc {from} {to} --no-color` where `{from}` is replaced with LESS file path while `{to}` is replaced with target CSS file path. Second one is `ts` that can be specified in `js` part of an asset bundle. The command that is run during conversion is in the same format that is used for `less`.

Error: not existing file: structure-extensions.md

Chapter 4

Handling Requests

Error: not existing file: runtime-bootstrapping.md

Error: not existing file: runtime-routing.md

Error: not existing file: runtime-requests.md

Error: not existing file: runtime-responses.md

Error: not existing file: runtime-sessions-cookies.md

4.1 URL Management

> Note: This section is under development.

The concept of URL management in Yii is fairly simple. URL management is based on the premise that the application uses internal routes and parameters everywhere. The framework itself will then translate routes into URLs, and vice versa, according to the URL manager's configuration. This approach allows you to change site-wide URLs merely by editing a single configuration file, without ever touching the application code.

4.1.1 Internal routes

When implementing an application using Yii, you'll deal with internal routes, often referred to as routes and parameters. Each controller and controller action has a corresponding internal route such as `site/index` or `user/create`. In the first example, `site` is referred to as the *controller ID* while `index` is referred to as the *action ID*. In the second example, `user` is the controller ID and `create` is the action ID. If the controller belongs to a *module*, the internal route is prefixed with the module ID. For example `blog/post/index` for a blog module (with `post` being the controller ID and `index` being the action ID).

4.1.2 Creating URLs

The most important rule for creating URLs in your site is to always do so using the URL manager. The URL manager is a built-in application component named `urlManager`. This component is accessible from both web and console applications via `\Yii::$app->urlManager`. The component makes available the two following URL creation methods:

- `createUrl($params)`
- `createAbsoluteUrl($params, $schema = null)`

The `createUrl` method creates an URL relative to the application root, such as `/index.php/site/index/`. The `createAbsoluteUrl` method creates an URL prefixed with the proper protocol and hostname: `http://www.example.com/index.php/site/index`. The former is suitable for internal application URLs, while the latter is used when you need to create URLs for external resources, such as connecting to third party services, sending email, generating RSS feeds etc.

Some examples:

```php
echo \Yii::$app->urlManager->createUrl(['site/page', 'id' => 'about']);
// /index.php/site/page/id/about/
echo \Yii::$app->urlManager->createUrl(['date-time/fast-forward', 'id' =>
    105])
```

```
// /index.php?r=date-time/fast-forward&id=105
echo \Yii::$app->urlManager->createAbsoluteUrl('blog/post/index');
// http://www.example.com/index.php/blog/post/index/
```

The exact format of the URL depends on how the URL manager is configured. The above examples may also output:

- `/site/page/id/about/`
- `/index.php?r=site/page&id=about`
- `/index.php?r=date-time/fast-forward&id=105`
- `/index.php/date-time/fast-forward?id=105`
- `http://www.example.com/blog/post/index/`
- `http://www.example.com/index.php?r=blog/post/index`

In order to simplify URL creation there is `yii\helpers\Url` helper. Assuming we're at `/index.php?r=management/default/users&id=10` the following is how `Url` helper works:

```
use yii\helpers\Url;

// currently active route
// /index.php?r=management/default/users
echo Url::to('');

// same controller, different action
// /index.php?r=management/default/page&id=contact
echo Url::toRoute(['page', 'id' => 'contact']);


// same module, different controller and action
// /index.php?r=management/post/index
echo Url::toRoute('post/index');

// absolute route no matter what controller is making this call
// /index.php?r=site/index
echo Url::toRoute('/site/index');

// url for the case sensitive action `actionHiTech` of the current
    controller
// /index.php?r=management/default/hi-tech
echo Url::toRoute('hi-tech');

// url for action the case sensitive controller, `DateTimeController::
    actionFastForward`
// /index.php?r=date-time/fast-forward&id=105
echo Url::toRoute(['/date-time/fast-forward', 'id' => 105]);

// get URL from alias
// http://google.com/
Yii::setAlias('@google', 'http://google.com/');
echo Url::to('@google');

// get home URL
// /index.php?r=site/index
echo Url::home();

Url::remember(); // save URL to be used later
Url::previous(); // get previously saved URL
```

> **Tip**: In order to generate URL with a hashtag, for example `/index.php?r=site/page&id=100#title`, you need to specify the parameter named `#` using `Url::to(['post/read', 'id' => 100, '#' => 'title'])`.

There's also `Url::canonical()` method that allows you to generate canonical URL[1] for the currently executing action. The method ignores all action parameters except ones passed via action arguments:

```
namespace app\controllers;

use yii\web\Controller;
use yii\helpers\Url;

class CanonicalController extends Controller
{
    public function actionTest($page)
    {
        echo Url::canonical();
    }
}
```

When accessed as `/index.php?r=canonical/test&page=hello&number=42` canonical URL will be `/index.php?r=canonical/test&page=hello`.

4.1.3 Customizing URLs

By default, Yii uses a query string format for URLs, such as `/index.php?r=news/view&id=100`. In order to make URLs human-friendly i.e., more readable, you need to configure the `urlManager` component in the application's configuration file. Enabling "pretty" URLs will convert the query string format to a directory-based format: `/index.php/news/view?id=100`. Disabling the `showScriptName` parameter means that `index.php` will not be part of the URLs. Here's the relevant part of the application's configuration file:

```
<?php
return [
    // ...
```

[1]https://en.wikipedia.org/wiki/Canonical_link_element

```
    'components' => [
        'urlManager' => [
            'enablePrettyUrl' => true,
            'showScriptName' => false,
        ],
    ],
];
```

Note that this configuration will only work if the web server has been properly configured for Yii, see installation.

Named parameters

A rule can be associated with a few `GET` parameters. These `GET` parameters appear in the rule's pattern as special tokens in the following format:

```
<ParameterName:ParameterPattern>
```

`ParameterName` is a name of a `GET` parameter, and the optional `ParameterPattern` is the regular expression that should be used to match the value of the `GET` parameter. In case `ParameterPattern` is omitted, it means the parameter should match any characters except `/`. When creating a URL, these parameter tokens will be replaced with the corresponding parameter values; when parsing a URL, the corresponding GET parameters will be populated with the parsed results.

Let's use some examples to explain how URL rules work. We assume that our rule set consists of three rules:

```
[
    'posts'=>'post/list',
    'post/<id:\d+>'=>'post/read',
    'post/<year:\d{4}>/<title>'=>'post/read',
]
```

- Calling `Url::toRoute('post/list')` generates `/index.php/posts`. The first rule is applied.
- Calling `Url::toRoute(['post/read', 'id' => 100])` generates `/index.php/post/100`. The second rule is applied.
- Calling `Url::toRoute(['post/read', 'year' => 2008, 'title' => 'a sample post'])` generates `/index.php/post/2008/a%20sample%20post`. The third rule is applied.
- Calling `Url::toRoute('post/read')` generates `/index.php/post/read`. None of the rules is applied, convention is used instead.

In summary, when using `createUrl` to generate a URL, the route and the `GET` parameters passed to the method are used to decide which URL rule to be applied. If every parameter associated with a rule can be found in the `GET` parameters passed to `createUrl`, and if the route of the rule also matches the route parameter, the rule will be used to generate the URL.

If the `GET` parameters passed to `Url::toRoute` are more than those required by a rule, the additional parameters will appear in the query string. For example, if we call `Url::toRoute(['post/read', 'id' => 100, 'year' => 2008])`, we will obtain `/index.php/post/100?year=2008`.

As we mentioned earlier, the other purpose of URL rules is to parse the requesting URLs. Naturally, this is an inverse process of URL creation. For example, when a user requests for `/index.php/post/100`, the second rule in the above example will apply, which resolves in the route `post/read` and the `GET` parameter `['id' => 100]` (accessible via `Yii::$app->request->get('id')`).

Parameterizing Routes

We may reference named parameters in the route part of a rule. This allows a rule to be applied to multiple routes based on matching criteria. It may also help reduce the number of rules needed for an application, and thus improve the overall performance.

We use the following example rules to illustrate how to parameterize routes with named parameters:

```
[
    '<controller:(post|comment)>/<id:\d+>/<action:(create|update|delete)>'
    => '<controller>/<action>',
    '<controller:(post|comment)>/<id:\d+>' => '<controller>/read',
    '<controller:(post|comment)>s' => '<controller>/list',
]
```

In the above example, we use two named parameters in the route part of the rules: `controller` and `action`. The former matches a controller ID to be either post or comment, while the latter matches an action ID to be create, update or delete. You may name the parameters differently as long as they do not conflict with GET parameters that may appear in URLs.

Using the above rules, the URL `/index.php/post/123/create` will be parsed as the route `post/create` with `GET` parameter `id=123`. Given the route `comment/list` and `GET` parameter `page=2`, we can create a URL `/index.php/comments?page=2`.

Parameterizing hostnames

It is also possible to include hostnames in the rules for parsing and creating URLs. One may extract part of the hostname to be a `GET` parameter. This is especially useful for handling subdomains. For example, the URL `http://admin.example.com/en/profile` may be parsed into GET parameters `user=admin` and `lang=en`. On the other hand, rules with hostname may also be used to create URLs with parameterized hostnames.

In order to use parameterized hostnames, simply declare URL rules with host info, e.g.:

```
[
    'http://<user:\w+>.example.com/<lang:\w+>/profile' => 'user/profile',
]
```

In the above example the first segment of the hostname is treated as the user parameter while the first segment of the path info is treated as the lang parameter. The rule corresponds to the `user/profile` route.

Note that `yii\web\UrlManager::showScriptName` will not take effect when a URL is being created using a rule with a parameterized hostname.

Also note that any rule with a parameterized hostname should NOT contain the subfolder if the application is under a subfolder of the Web root. For example, if the application is under `http://www.example.com/sandbox/blog`, then we should still use the same URL rule as described above without the subfolder `sandbox/blog`.

Faking URL Suffix

```
<?php
return [
    // ...
    'components' => [
        'urlManager' => [
            'suffix' => '.html',
        ],
    ],
];
```

Handling REST requests

TBD: - RESTful routing: `yii\filters\VerbFilter`, `yii\web\UrlManager::$rules` - Json API: - response: `yii\web\Response::format` - request: `yii\web\Request::$parsers`, `yii\web\JsonParser`

4.1.4 URL parsing

Complimentary to creating URLs Yii also handles transforming custom URLs back into internal routes and parameters.

Strict URL parsing

By default if there's no custom rule for a URL and the URL matches the default format such as `/site/page`, Yii tries to run the corresponding controller's action. This behavior can be disabled so if there's no custom rule match, a 404 not found error will be produced immediately.

```
<?php
return [
    // ...
    'components' => [
        'urlManager' => [
            'enableStrictParsing' => true,
        ],
    ],
];
```

4.1.5 Creating your own rule classes

`yii\web\UrlRule` class is used for both parsing URL into parameters and creating URL based on parameters. Despite the fact that default implementation is flexible enough for the majority of projects, there are situations when using your own rule class is the best choice. For example, in a car dealer website, we may want to support the URL format like `/Manufacturer/Model`, where `Manufacturer` and `Model` must both match some data in a database table. The default rule class will not work because it mostly relies on statically declared regular expressions which have no database knowledge.

We can write a new URL rule class by extending from `yii\web\UrlRule` and use it in one or multiple URL rules. Using the above car dealer website as an example, we may declare the following URL rules in application config:

```
// ...
'components' => [
    'urlManager' => [
        'rules' => [
            '<action:(login|logout|about)>' => 'site/<action>',

            // ...

            ['class' => 'app\components\CarUrlRule', 'connectionID' => 'db',
     /* ... */],
        ],
    ],
],
```

In the above, we use the custom URL rule class `CarUrlRule` to handle the URL format `/Manufacturer/Model`. The class can be written like the following:

```
namespace app\components;

use yii\web\UrlRule;

class CarUrlRule extends UrlRule
{
    public $connectionID = 'db';

    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
    }
}
```

Besides the above usage, custom URL rule classes can also be implemented for many other purposes. For example, we can write a rule class to log the URL parsing and creation requests. This may be useful during development stage. We can also write a rule class to display a special 404 error page in case all other URL rules fail to resolve the current request. Note that in this case, the rule of this special class must be declared as the last rule.

4.2 Error Handling

Note: This section is under development.

Error handling in Yii is different than handling errors in plain PHP. First of all, Yii will convert all non-fatal errors to *exceptions*:

```
use yii\base\ErrorException;
use Yii;

try {
    10/0;
} catch (ErrorException $e) {
    Yii::warning("Tried dividing by zero.");
}

// execution may continue
```

As demonstrated above you may handle errors using `try-catch`.

Second, even fatal errors in Yii are rendered in a nice way. This means that in debugging mode, you can trace the causes of fatal errors in order to more quickly identify the cause of the problem.

4.2.1 Rendering errors in a dedicated controller action

The default Yii error page is great when developing a site, and is acceptable for production sites if `YII_DEBUG` is turned off in your bootstrap `index.php` file. But you may want to customize the default error page to make it more suitable for your project.

The easiest way to create a custom error page it is to use a dedicated controller action for error rendering. First, you'll need to configure the `errorHandler` component in the application's configuration:

```
// ...
'components' => [
    // ...
    'errorHandler' => [
        'errorAction' => 'site/error',
    ],
]
```

With that configuration in place, whenever an error occurs, Yii will execute the `error`-action of the `site`-controller. That action should look for an exception and, if present, render the proper view file, passing along the exception:

```
public function actionError()
{
    $exception = \Yii::$app->errorHandler->exception;
    if ($exception !== null) {
        return $this->render('error', ['exception' => $exception]);
    }
}
```

Next, you would create the `views/site/error.php` file, which would make use of the exception. The exception object has the following properties:

- `statusCode`: the HTTP status code (e.g. 403, 500). Available for `yii\web\HttpException` only.
- `code`: the code of the exception.
- `message`: the error message.
- `file`: the name of the PHP script file where the error occurs.
- `line`: the line number of the code where the error occurs.
- `trace`: the call stack of the error.

4.2.2 Rendering errors without a dedicated controller action

Instead of creating a dedicated action within the Site controller, you could just indicate to Yii what class should be used to handle errors:

```
public function actions()
{
    return [
        'error' => [
            'class' => 'yii\web\ErrorAction',
        ],
    ];
}
```

After associating the class with the error as in the above, define the `views/site/error.php` file, which will automatically be used. The view will be passed three variables:

- `$name`: the error name
- `$message`: the error message
- `$exception`: the exception being handled

The `$exception` object will have the same properties as outlined above.

4.3 Logging

> Note: This section is under development.

Yii provides flexible and extensible logger that is able to handle messages according to severity level or their type. You may filter messages by multiple criteria and forward them to files, email, debugger etc.

4.3.1 Logging basics

Basic logging is as simple as calling one method:

```
\Yii::info('Hello, I am a test log message');
```

You can log simple strings as well as more complex data structures such as arrays or objects. When logging data that is not a string the default yii log targets will serialize the value using `yii\helpers\Vardumper::export()`.

Message category

Additionally to the message itself message category could be specified in order to allow filtering such messages and handing these differently. Message category is passed as a second argument of logging methods and is `application` by default.

Severity levels

There are multiple severity levels and corresponding methods available:

- `Yii::trace` used maily for development purpose to indicate workflow of some code. Note that it only works in development mode when `YII_DEBUG` is set to `true`.
- `Yii::error` used when there's unrecoverable error.
- `Yii::warning` used when an error occurred but execution can be continued.
- `Yii::info` used to keep record of important events such as administrator logins.

4.3.2 Log targets

When one of the logging methods is called, message is passed to `yii\log\Logger` component accessible as `Yii::getLogger()`. Logger accumulates messages in memory and then when there are enough messages or when the current request finishes, sends them to different log targets, such as file or email.

You may configure the targets in application configuration, like the following:

```
[
    'components' => [
        'log' => [
            'targets' => [
                'file' => [
                    'class' => 'yii\log\FileTarget',
                    'levels' => ['trace', 'info'],
                    'categories' => ['yii\*'],
                ],
                'email' => [
                    'class' => 'yii\log\EmailTarget',
                    'levels' => ['error', 'warning'],
                    'message' => [
                        'to' => ['admin@example.com', 'developer@example.com
    '],
                        'subject' => 'New example.com log message',
                    ],
                ],
            ],
        ],
    ],
]
```

In the config above we are defining two log targets: `yii\log\FileTarget` and `yii\log\EmailTarget`. In both cases we are filtering messages handles by these targets by severity. In case of file target we're additionally filter by category. `yii\*` means all categories starting with `yii\`.

Each log target can have a name and can be referenced via the `yii\log\Logger::targets` property as follows:

```
Yii::$app->log->targets['file']->enabled = false;
```

When the application ends or `yii\log\Logger::flushInterval` is reached, Logger will call `yii\log\Logger::flush()` to send logged messages to different log targets, such as file, email, web.

4.3.3 Profiling

Performance profiling is a special type of message logging that can be used to measure the time needed for the specified code blocks to execute and find out what the performance bottleneck is.

To use it we need to identify which code blocks need to be profiled. Then we mark the beginning and the end of each code block by inserting the following methods:

```
\Yii::beginProfile('myBenchmark');
...code block being profiled...
\Yii::endProfile('myBenchmark');
```

where `myBenchmark` uniquely identifies the code block.

Note, code blocks need to be nested properly such as

```
\Yii::beginProfile('block1');
    // some code to be profiled
    \Yii::beginProfile('block2');
        // some other code to be profiled
    \Yii::endProfile('block2');
\Yii::endProfile('block1');
```

Profiling results could be displayed in debugger.

Chapter 5

Key Concepts

5.1 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`, 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 events and behaviors 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 the 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:

```
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 guideliness 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 due to the fact that 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()`. 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 an object instance, it has already been initialized to a proper state that you can reliably work with.

5.2 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 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 have to call `trim()` everywhere in your code where you met set the `label` property. If in the future, the `label` property gets a new requirement, such as the first letter must be captialized, 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 such support, it should extend from `yii\base\Object` or 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 extend 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 refer to a private attributed named `_label`.)

Properties defined by getters and setters can be used like class member variables. The main difference is that when such a 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`. 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 for the visibility of a property if the defining getter or setter method 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 this same manner.

Returning back to the problem described at the beginning of this guide, instead of calling `trim()` everywhere a `label` value is assigned, `trim()` only needs to be invoked within the setter `setLabel()`. And if a new requirement comes that requires 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`.

5.3 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 a child class.

5.3.1 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);
    }
}
```

In the above code, when you call `bar()`, it 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` is used to represent `hello`. This approach has two benefits. First, it prevents typos and can impact IDE auto-completion support. Second, you can tell what events are supported by a class by simply checking the 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 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 handlers that are attached to the named event.

5.3.2 Event Handlers

An event handler is a PHP callback[1] that gets executed when the event it is attached to is triggered. You can use one of the following callbacks:

- a global PHP function specified in terms of a string, e.g., `'trim()'`;
- an object method specified in terms of an array of an object and a method name, e.g., `[$object, $method]`;
- a static class method specified in terms of an array of a class name and a method name, e.g., `[$class, $method]`;
- an anonymous function, e.g., `function ($event) { ... }`.

The signature of an event handler is:

```
function ($event) {
    // $event is an object of yii\base\Event or its child class
}
```

Through the `$event` parameter, an event handler may get the following information about an event:

- `yii\base\Event::name`
- `yii\base\Event::sender`: the object whose `trigger()` method is called.
- `yii\base\Event::data`: the data that is provided when attaching the event handler (to be explained shortly).

5.3.3 Attaching Event Handlers

You can attach a handler to an event by calling the `yii\base\Component::on()` method. For example,

```
$foo = new Foo;

// the handler is a global function
$foo->on(Foo::EVENT_HELLO, 'function_name');

// the handler is an object method
$foo->on(Foo::EVENT_HELLO, [$object, 'methodName']);

// the handler is a static class method
$foo->on(Foo::EVENT_HELLO, ['app\components\Bar', 'methodName']);
```

[1] http://www.php.net/manual/en/language.types.callable.php

```
// the 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 to "on"
$foo->on(Foo::EVENT_HELLO, function ($event) {
    echo $event->data;
}, 'abc');
```

You may attach one or multiple handlers to a single event. When an event is triggered, the attached handlers will be called in the order they are attached to the event. If a handler needs to stop the invocation of the handlers behind it, it may set the `yii\base\Event::handled` property of the `$event` parameter to be true, like the following,

```
$foo->on(Foo::EVENT_HELLO, function ($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, y ou may call `yii\base\Component::on()` by passing the fourth parameter `$append` as false:

```
$foo->on(Foo::EVENT_HELLO, function ($event) {
    // ...
}, $data, false);
```

5.3.4 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, we assume 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);
```

5.3.5 Class-Level Event Handlers

In the above subsections, we have described how to attach a handler to an event at *instance level*. Sometimes, you may want to respond to an event triggered by EVERY instance of a class instead of a specific instance. Instead of attaching an event handler to every instance, you may attach the handler at *class level* by calling the static method `yii\base\Event::on()`.

For example, an Active Record object will trigger a `yii\base\ActiveRecord::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 write the following code:

```
use Yii;
use yii\base\Event;
use yii\db\ActiveRecord;

Event::on(ActiveRecord::className(), ActiveRecord::EVENT_AFTER_INSERT,
    function ($event) {
    Yii::trace(get_class($event->sender) . ' is inserted');
});
```

The event handler will get invoked whenever an instance of `yii\base\ActiveRecord` or its child class triggers the `yii\base\ActiveRecord::EVENT_AFTER_INSERT` event. In the handler, you can get the object that triggers the event through `$event->sender`.

When an object triggers an event, it will first call instance-level handlers, followed by class-level handlers.

You may trigger an *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) {
    echo $event->sender;  // displays "app\models\Foo"
});
```

```
Event::trigger(Foo::className(), Foo::EVENT_HELLO);
```

Note that in this case, `$event->sender` refers to the name of the class triggering the event instead of an object instance.

> Note: Because a class-level handler will respond to an event triggered by any instance of that class or its child class, 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);
```

5.3.6 Global Events

The so-called *global event* is actually a trick based on the event mechanism described above. It requires a globally accessible singleton, such as the application instance.

An event sender, instead of calling its own `trigger()` method, will call the singleton's `trigger()` method to trigger the event. Similarly, the event handlers are attached to the event of 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 global events is that you do not need the 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").

5.4 Behaviors

Behaviors are instances of `yii\base\Behavior` or its child class. Behaviors, also known as mixins[2], allow you to enhance the functionality of an existing `yii\base\Component` class without the need of changing its class inheritance. When a behavior is attached to a component, it will "inject" its methods and properties into the component, and you can access these methods and properties as if they are defined by the component class. Moreover, a behavior can respond to the events triggered by the component so that it can customize or adapt the normal code execution of the component.

5.4.1 Using Behaviors

To use a behavior, you first need to attach it to a `yii\base\Component`. We will describe how to attach a behavior in the next subsection.

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, like the following,

```
// "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,

```
// bar() is a public method defined in the behavior class
$component->bar();
```

As you can see, although `$component` does not define `prop1` and `bar()`, they can be used as if they are part of the component definition.

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 being 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, like the following,

```
$behavior = $component->getBehavior('myBehavior');
```

You may also get all behaviors attached to a component:

```
$behaviors = $component->getBehaviors();
```

5.4.2 Attaching Behaviors

You can attach a behavior to a `yii\base\Component` either statically or dynamically. The former is more commonly used in practice.

[2]http://en.wikipedia.org/wiki/Mixin

To attach a behavior statically, override the `yii\base\Component::behaviors()` method of the component class that it is being attached. For example,

```
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',
            ]
        ];
    }
}
```

The `yii\base\Component::behaviors()` method should return a list of behavior configurations. Each behavior configuration can be either a behavior class name or a configuration array.

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 that it is attached to. For example,

```
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 also attach behaviors through configurations. For more details, please refer to the Configurations section.

5.4.3 Detaching Behaviors

To detach a behavior, you can call `yii\base\Component::detachBehavior()` with the name associated with the behavior:

```
$component->detachBehavior('myBehavior1');
```

You may also detach *all* behaviors:

```
$component->detachBehaviors();
```

5.4.4 Defining Behaviors

To define a behavior, create a class by extending from `yii\base\Behavior` or its child class. For example,

```
namespace app\components;

use yii\base\Model;
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` which will provide two properties `prop1` and `prop2`, and one method `foo()` to the component it is attached to. Note that property `prop2` is defined via the getter `getProp2()` and the setter `setProp2()`. This is so because `yii\base\Object` is an ancestor class of `yii\base\Behavior`, which supports defining properties by getters/setters.

Within a behavior, you can access the component that the behavior is attached to through the `yii\base\Behavior::owner` property.

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()` method should return a list of events and their corresponding handlers. The above example declares that the `yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE` event and 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, 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) {
}
```

5.4.5 Using `TimestampBehavior`

To wrap up, let's take a look at `yii\behaviors\TimestampBehavior` - a behavior that supports automatically updating the timestamp attributes of an `yii\db\ActiveRecord` when it is being saved.

First, attach this behavior to the `yii\db\ActiveRecord` 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'],
                ],
            ],
        ];
    }
}
```

The behavior configuration above specifies that

- when the record is being inserted, the behavior should assign the current timestamp to the `created_at` and `updated_at` attributes;
- when the record is being updated, the behavior should assign the current timestamp to the `updated_at` attribute.

Now 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 timestamp:

```
$user = new User;
$user->email = 'test@example.com';
$user->save();
echo $user->created_at;  // shows the current timestamp
```

The `yii\behaviors\TimestampBehavior` also offers a useful method `yii\behaviors\TimestampBehavior::touch()` which will assign the current timestamp to a specified attribute and save it to the database:

```
$user->touch('login_time');
```

5.4.6 Comparison with Traits

While behaviors are similar to traits[3] 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 rather than replacements to each other.

Pros for 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 you to modify the component class. To use a trait, you must modify 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 is name conflict among different behaviors attached to the same component, the conflict is automatically resolved by respecting the behavior that is attached to the component first. Name conflict caused by different traits requires you to manually resolve it by renaming the affected properties or methods.

Pros for Traits

Traits are much more efficient than behaviors because behaviors are objects which take both time and memory.

IDEs are more friendly to traits as they are language construct.

5.5 Configurations

Configurations are widely used in Yii for creating new objects or initializing existing objects. They usually include the class names of the objects being created and a list of initial values that should be assigned to object properties. They may also include a list of handlers that should be attached to the object events, and/or a list of behaviors that should be attached to the objects.

[3] `http://www.php.net/traits`

In the following, a configuration is used to create and initialize a DB 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 and creates an object based on the class name specified in the configuration. When the object is being instantiated, the rest of the configuration will be used to initialize the object properties, event handlers and/or behaviors.

If you already have an object, you may use `Yii::configure()` to initialize the object properties with a configuration, like the following,

```
Yii::configure($object, $config);
```

Note that in this case, the configuration should not contain the `class` element.

5.5.1 Configuration Format

The format of a configuration can be formally described as follows,

```
[
    '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 property initial values. 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 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.
- And 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 `on .` `$behaviorConfig` represents the configuration for creating a behavior, like a normal configuration as we are describing here.

Below is an example showing a configuration with property initial 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 ...
    ],
]
```

5.5.2 Using Configurations

Configurations are used in many places in Yii. At the beginning of this section, we have shown how to use 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

Configuration for an application is probably one of the most complex configurations. This is because the `yii\web\Application` class has a lot of configurable properties and events. More importantly, its `yii\web\Application ::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 application template.

```
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'extensions' => require(__DIR__ . '/../vendor/yiisoft/extensions.php'),
    'components' => [
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'mail' => [
            '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();
```

For more details about configuring the `components` property of an application can be found in the Applications section and the Service Locator section.

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::beginWidget()` 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->isGuest],
    ],
]);
```

The above code creates a `Menu` widget and initializes its `activeItems` 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.

5.5.3 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',
    ],
    'mail' => [
        '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();
```

5.5.4 Default Configurations

The `Yii::createObject()` method is implemented based on a dependency injection container. It allows you specify a set of the so-called *default configurations* which will be applied to ANY 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.

5.5.5 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;
```

5.6 Aliases

Aliases are used to represent file paths or URLs to avoid hard-coding absolute paths or URLs in your code. An alias must start with a `@` character so that it can be differentiated from file paths and URLs. For example, the alias `@yii` represents the installation path of the Yii framework, while `@web` represents the base URL for the currently running Web application.

5.6.1 Defining Aliases

You can call `Yii::setAlias()` to define an alias for a given file path or URL. For example,

```
// an alias of file path
Yii::setAlias('@foo', '/path/to/foo');

// an alias of URL
Yii::setAlias('@bar', 'http://www.example.com');
```

> Note: A file path or URL being aliased may NOT necessarily refer to an existing file or resource.

Given an alias, you may derive a new alias (without the need of calling `Yii::setAlias()`) by appending a slash `/` followed with one or several path segments. We call the aliases defined via `Yii::setAlias()` *root aliases*, while the aliases derived from them *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 alias or derived alias is fine):

```
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, like the following,

```
return [
    // ...
    'aliases' => [
        '@foo' => '/path/to/foo',
        '@bar' => 'http://www.example.com',
    ],
];
```

5.6.2 Resolving Aliases

You can call `Yii::getAlias()` to resolve a root alias into the file path or URL it is representing. The same method can also resolve a derived alias into the corresponding file path or URL. For example,

```
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. For example,

```
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`.

5.6.3 Using Aliases

Aliases are recognized in many places in Yii without the need of calling `Yii::getAlias()` to convert them into paths/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.

5.6.4 Predefined Aliases

Yii predefines a set of aliases to ease the need of referencing commonly used file paths and URLs. The following is the list of the predefined aliases:

- `@yii`: the directory where the `BaseYii.php` file is located (also called the framework directory).

- `@app`: the `yii\base\Application::basePath` of the currently running application.
- `@runtime`: the `yii\base\Application::runtimePath` of the currently running application.
- `@vendor`: the [[yii\base\Application::vendorPath|Composer vendor directory].
- `@webroot`: the Web root directory of the currently running Web application.
- `@web`: the base URL of the currently running Web application.

The `@yii` alias is defined when you include the `Yii.php` file in your entry script, while the rest of the aliases are defined in the application constructor when applying the application configuration.

5.6.5 Extension Aliases

An alias is automatically defined for each extension that is installed via Composer. The alias is named after the root namespace of the extension as declared in its `composer.json` file, and it 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:

```
Yii::setAlias('@yii/jui', 'VendorPath/yiisoft/yii2-jui');
```

5.7 Class Autoloading

Yii relies on the class autoloading mechanism[4] to locate and include required class files. It provides a high-performance class autoloader that is compliant to the PSR-4 standard[5]. 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.

[4]`http://www.php.net/manual/en/language.oop5.autoload.php`
[5]`https://github.com/php-fig/fig-standards/blob/master/proposed/psr-4-autoloader/psr-4-autoloader.md`

5.7.1 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 some 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 with the leading backslash
$classFile = Yii::getAlias('@' . str_replace('\\', '/', $className) . '.php'
    );
```

For example, if a class name 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 able to be resolved into a file path, either `@foo` or `@foo/bar` must be a root alias.

When you are using the Basic Application 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 we just described.

In the Advanced Application Template, each tier has its own root alias. For example, the front-end tier has a root alias `@frontend` while the back-end tier `@backend`. As a result, you may put the front-end classes under the namespace `frontend` while the back-end classes under `backend`. This will allow these classes to be autoloaded by the Yii autoloader.

5.7.2 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 check. This makes class autoloading super fast. In fact, all core Yii classes are being autoloaded this way.

You may add a class to the class map `Yii::$classMap` as follows,

```
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.

5.7.3 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 some 3rd-party libraries that have their autoloaders, you should also install them.

When you are 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 to be the first one responding to any class autoloading request. For example, the following code is extracted from the entry script of the Basic Application 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.

5.7.4 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[6] 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.

5.8 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 has only a single instance which is 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 its 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

[6]https://getcomposer.org/doc/04-schema.md#autoload

application components, such as the `request`, `response`, `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. 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 "db" 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 is registered, you can access it using its ID in one of the following two 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()` is provided so that you can configure it and register multiple components at once. The following code shows a configuration array that can be used to configure an application and register 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 () {
            return new app\components\SolrService;
        },
    ],
];
```

5.9 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's article[7] has well explained why DI container is useful. Here we will mainly explain the usage of the DI container provided by Yii.

5.9.1 Dependency Injection

Yii provides the DI container feature through the class `yii\di\Container`. It supports the following kinds of dependency injection:

- Constructor 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

[7] `http://martinfowler.com/articles/injection.html`

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);
```

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'),
]);
```

PHP Callable Injection

In this case, the container will use a registered PHP callable to build new instances of a class. The callable is responsible to resolve the dependencies and inject them appropriately to the newly created objects. For example,

```
$container->set('Foo', function () {
    return new Foo(new Bar);
});

$foo = $container->get('Foo');
```

5.9.2 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',
]);
```

5.9.3 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()` to create new objects. The method takes a dependency name, which can be a class name, an interface name or an alias name. The dependency name may or may not be registered via `set()` or `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, ['type
    ' => 1]);
$engine = $container->get('app\components\SearchEngine', [$apiKey], ['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')`.

```
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);
```

5.9.4 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::$container`. When you call `Yii::createObject()`, the method will actually call the container's `yii\di\Container::get()` method to create a new object. As aforementioned, the DI container will automatically resolve the dependencies (if any) and inject them into the newly created object. Because Yii uses `Yii::createObject()` in most of its core code to create new objects, this means you can customize the objects globally by dealing with `Yii::$container`.

For example, you can 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]);
```

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, $config = [])
    {
        $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\
    BookingService');
```

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.

5.9.5 When to Register Dependencies

Because dependencies are needed when new objects are being created, their registration should be done as early as possible. The followings are the recommended practices:

- If you are the developer of an application, you can register dependencies in your application's entry script or in a script that is included by the entry script.
- If you are the developer of a redistributable extension, you can register dependencies in the bootstrap class of the extension.

5.9.6 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[8] 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.

[8] `http://martinfowler.com/articles/injection.html`

Chapter 6

Working with Databases

6.1 Database basics

> Note: This section is under development.

Yii has a database access layer built on top of PHP's PDO[1]. It provides uniform API and solves some inconsistencies between different DBMS. By default Yii supports the following DBMS:

- MySQL[2]
- MariaDB[3]
- SQLite[4]
- PostgreSQL[5]
- CUBRID[6]: version 9.1.0 or higher.
- Oracle[7]
- MSSQL[8]: version 2012 or above is required if you want to use LIMIT/OFFSET.

[1]`http://www.php.net/manual/en/book.pdo.php`
[2]`http://www.mysql.com/`
[3]`https://mariadb.com/`
[4]`http://sqlite.org/`
[5]`http://www.postgresql.org/`
[6]`http://www.cubrid.org/`
[7]`http://www.oracle.com/us/products/database/overview/index.html`
[8]`https://www.microsoft.com/en-us/sqlserver/default.aspx`

6.1.1 Configuration

In order to start using database you need to configure database connection component first by adding `db` component to application configuration (for "basic" web application it's `config/web.php`) like the following:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=mydatabase', // MySQL,
    MariaDB
            //'dsn' => 'sqlite:/path/to/database/file', // SQLite
            //'dsn' => 'pgsql:host=localhost;port=5432;dbname=mydatabase',
    // PostgreSQL
            //'dsn' => 'cubrid:dbname=demodb;host=localhost;port=33000', //
    CUBRID
            //'dsn' => 'sqlsrv:Server=localhost;Database=mydatabase', // MS
    SQL Server, sqlsrv driver
            //'dsn' => 'dblib:host=localhost;dbname=mydatabase', // MS SQL
    Server, dblib driver
            //'dsn' => 'mssql:host=localhost;dbname=mydatabase', // MS SQL
    Server, mssql driver
            //'dsn' => 'oci:dbname=//localhost:1521/mydatabase', // Oracle
            'username' => 'root',
            'password' => '',
            'charset' => 'utf8',
        ],
    ],
    // ...
];
```

There is a peculiarity when you want to work with the database through the `ODBC` layer. When using `ODBC`, connection `DSN` doesn't indicate uniquely what database type is being used. That's why you have to override `driverName` property of `yii\db\Connection` class to disambiguate that:

```
'db' => [
    'class' => 'yii\db\Connection',
    'driverName' => 'mysql',
    'dsn' => 'odbc:Driver={MySQL};Server=localhost;Database=test',
    'username' => 'root',
    'password' => '',
],
```

Please refer to the PHP manual[9] for more details on the format of the DSN string.

After the connection component is configured you can access it using the following syntax:

[9] `http://www.php.net/manual/en/function.PDO-construct.php`

```
$connection = \Yii::$app->db;
```

You can refer to `yii\db\Connection` for a list of properties you can configure. Also note that you can define more than one connection component and use both at the same time if needed:

```
$primaryConnection = \Yii::$app->db;
$secondaryConnection = \Yii::$app->secondDb;
```

If you don't want to define the connection as an application component you can instantiate it directly:

```
$connection = new \yii\db\Connection([
    'dsn' => $dsn,
     'username' => $username,
     'password' => $password,
]);
$connection->open();
```

> **Tip**: if you need to execute additional SQL queries right after establishing a connection you can add the following to your application configuration file:
>
> ```
> return [
>     // ...
>     'components' => [
>         // ...
>         'db' => [
>             'class' => 'yii\db\Connection',
>             // ...
>             'on afterOpen' => function($event) {
>                 $event->sender->createCommand("SET time_zone = '
>     UTC'")->execute();
>             }
>         ],
>     ],
>     // ...
> ];
> ```

6.1.2 Basic SQL queries

Once you have a connection instance you can execute SQL queries using `yii\db\Command`.

SELECT

When query returns a set of rows:

```
$command = $connection->createCommand('SELECT * FROM post');
$posts = $command->queryAll();
```

When only a single row is returned:

```
$command = $connection->createCommand('SELECT * FROM post WHERE id=1');
$post = $command->queryOne();
```

When there are multiple values from the same column:

```
$command = $connection->createCommand('SELECT title FROM post');
$titles = $command->queryColumn();
```

When there's a scalar value:

```
$command = $connection->createCommand('SELECT COUNT(*) FROM post');
$postCount = $command->queryScalar();
```

UPDATE, INSERT, DELETE etc.

If SQL executed doesn't return any data you can use command's `execute` method:

```
$command = $connection->createCommand('UPDATE post SET status=1 WHERE id=1')
    ;
$command->execute();
```

Alternatively the following syntax that takes care of proper table and column names quoting is possible:

```
// INSERT
$connection->createCommand()->insert('user', [
    'name' => 'Sam',
    'age' => 30,
])->execute();

// INSERT multiple rows at once
$connection->createCommand()->batchInsert('user', ['name', 'age'], [
    ['Tom', 30],
    ['Jane', 20],
    ['Linda', 25],
])->execute();

// UPDATE
$connection->createCommand()->update('user', ['status' => 1], 'age > 30')->
    execute();

// DELETE
$connection->createCommand()->delete('user', 'status = 0')->execute();
```

6.1.3 Quoting table and column names

Most of the time you would use the following syntax for quoting table and column names:

```
$sql = "SELECT COUNT([[$column]]) FROM {{table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

In the code above `[[X]]` will be converted to properly quoted column name while `{{Y}}` will be converted to properly quoted table name.

For table names there's a special variant `{{%Y}}` that allows you to automatically appending table prefix if it is set:

```
$sql = "SELECT COUNT([[$column]]) FROM {{%table}}";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

The code above will result in selecting from `tbl_table` if you have table prefix configured like the following in your config file:

```
return [
    // ...
    'components' => [
        // ...
        'db' => [
            // ...
            'tablePrefix' => 'tbl_',
        ],
    ],
];
```

The alternative is to quote table and column names manually using `yii\db\Connection::quoteTableName()` and `yii\db\Connection::quoteColumnName()`:

```
$column = $connection->quoteColumnName($column);
$table = $connection->quoteTableName($table);
$sql = "SELECT COUNT($column) FROM $table";
$rowCount = $connection->createCommand($sql)->queryScalar();
```

6.1.4 Prepared statements

In order to securely pass query parameters you can use prepared statements:

```
$command = $connection->createCommand('SELECT * FROM post WHERE id=:id');
$command->bindValue(':id', $_GET['id']);
$post = $command->query();
```

Another usage is performing a query multiple times while preparing it only once:

```
$command = $connection->createCommand('DELETE FROM post WHERE id=:id');
$command->bindParam(':id', $id);

$id = 1;
$command->execute();

$id = 2;
$command->execute();
```

6.1.5 Transactions

You can perform transactional SQL queries like the following:

```
$transaction = $connection->beginTransaction();
try {
    $connection->createCommand($sql1)->execute();
     $connection->createCommand($sql2)->execute();
    // ... executing other SQL statements ...
    $transaction->commit();
} catch(Exception $e) {
    $transaction->rollBack();
}
```

You can also nest multiple transactions, if needed:

```
// outer transaction
$transaction1 = $connection->beginTransaction();
try {
    $connection->createCommand($sql1)->execute();

    // inner transaction
    $transaction2 = $connection->beginTransaction();
    try {
        $connection->createCommand($sql2)->execute();
        $transaction2->commit();
    } catch (Exception $e) {
        $transaction2->rollBack();
    }

    $transaction1->commit();
} catch (Exception $e) {
    $transaction1->rollBack();
}
```

6.1.6 Working with database schema

Getting schema information

You can get a `yii\db\Schema` instance like the following:

```
$schema = $connection->getSchema();
```

It contains a set of methods allowing you to retrieve various information about the database:

```
$tables = $schema->getTableNames();
```

For the full reference check `yii\db\Schema`.

Modifying schema

Aside from basic SQL queries `yii\db\Command` contains a set of methods allowing to modify database schema:

- createTable, renameTable, dropTable, truncateTable
- addColumn, renameColumn, dropColumn, alterColumn

- addPrimaryKey, dropPrimaryKey
- addForeignKey, dropForeignKey
- createIndex, dropIndex

These can be used as follows:

```
// CREATE TABLE
$connection->createCommand()->createTable('post', [
    'id' => 'pk',
    'title' => 'string',
    'text' => 'text',
]);
```

For the full reference check `yii\db\Command`.

6.2 Query Builder and Query

> Note: This section is under development.

Yii provides a basic database access layer as described in the Database basics section. The database access layer provides a low-level way to interact with the database. While useful in some situations, it can be tedious and error-prone to write raw SQLs. An alternative approach is to use the Query Builder. The Query Builder provides an object-oriented vehicle for generating queries to be executed.

A typical usage of the query builder looks like the following:

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

// which is equivalent to the following code:

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

// Create a command. You can get the actual SQL using $command->sql
$command = $query->createCommand();

// Execute the command:
$rows = $command->queryAll();
```

6.2.1 Query Methods

As you can see, `yii\db\Query` is the main player that you need to deal with. Behind the scene, `Query` is actually only responsible for representing various query information. The actual query building logic is done by `yii\db\QueryBuilder` when you call the `createCommand()` method, and the query execution is done by `yii\db\Command`.

For convenience, `yii\db\Query` provides a set of commonly used query methods that will build the query, execute it, and return the result. For example,

- `yii\db\Query::all()`: builds the query, executes it and returns all results as an array.
- `yii\db\Query::one()`: returns the first row of the result.
- `yii\db\Query::column()`: returns the first column of the result.
- `yii\db\Query::scalar()`: returns the first column in the first row of the result.
- `yii\db\Query::exists()`: returns a value indicating whether the query results in anything.
- `yii\db\Query::count()`: returns the result of a `COUNT` query. Other similar methods include `sum($q)`, `average($q)`, `max($q)`, `min($q)`, which support the so-called aggregational data query. `$q` parameter is mandatory for these methods and can be either the column name or expression.

6.2.2 Building Query

In the following, we will explain how to build various clauses in a SQL statement. For simplicity, we use `$query` to represent a `yii\db\Query` object.

SELECT

In order to form a basic `SELECT` query, you need to specify what columns to select and from what table:

```
$query->select('id, name')
    ->from('user');
```

Select options can be specified as a comma-separated string, as in the above, or as an array. The array syntax is especially useful when forming the selection dynamically:

```
$query->select(['id', 'name'])
    ->from('user');
```

> Info: You should always use the array format if your `SELECT` clause contains SQL expressions. This is because a SQL expression like `CONCAT(first_name, last_name) AS full_name` may contain commas. If you list it together with other columns in a string, the expression may be split into several parts by commas, which is not what you want to see.

When specifying columns, you may include the table prefixes or column aliases, e.g., `user.id`, `user.id AS user_id`. If you are using array to specify the columns, you may also use the array keys to specify the column aliases, e.g., `['user_id' => 'user.id', 'user_name' => 'user.name']`.

To select distinct rows, you may call `distinct()`, like the following:

```
$query->select('user_id')->distinct()->from('post');
```

FROM

To specify which table(s) to select data from, call `from()`:

```
$query->select('*')->from('user');
```

You may specify multiple tables using a comma-separated string or an array. Table names can contain schema prefixes (e.g. `'public.user'`) and/or table aliases (e.g. `'user u'`). The method will automatically quote the table names unless it contains some parenthesis (which means the table is given as a sub-query or DB expression). For example,

```
$query->select('u.*, p.*')->from(['user u', 'post p']);
```

When the tables are specified as an array, you may also use the array keys as the table aliases (if a table does not need alias, do not use a string key). For example,

```
$query->select('u.*, p.*')->from(['u' => 'user', 'p' => 'post']);
```

You may specify a sub-query using a `Query` object. In this case, the corresponding array key will be used as the alias for the sub-query.

```
$subQuery = (new Query())->select('id')->from('user')->where('status=1');
$query->select('*')->from(['u' => $subQuery]);
```

WHERE

Usually data is selected based upon certain criteria. Query Builder has some useful methods to specify these, the most powerful of which being `where`. It can be used in multiple ways.

The simplest way to apply a condition is to use a string:

```
$query->where('status=:status', [':status' => $status]);
```

When using strings, make sure you're binding the query parameters, not creating a query by string concatenation. The above approach is safe to use, the following is not:

```
$query->where("status=$status"); // Dangerous!
```

Instead of binding the status value immediately, you can do so using `params` or `addParams`:

```
$query->where('status=:status');
$query->addParams([':status' => $status]);
```

Multiple conditions can simultaneously be set in `where` using the *hash format*:

```
$query->where([
    'status' => 10,
    'type' => 2,
    'id' => [4, 8, 15, 16, 23, 42],
]);
```

That code will generate the following SQL:

```
WHERE (`status` = 10) AND (`type` = 2) AND (`id` IN (4, 8, 15, 16, 23, 42))
```

NULL is a special value in databases, and is handled smartly by the Query Builder. This code:

```
$query->where(['status' => null]);
```

results in this WHERE clause:

```
WHERE (`status` IS NULL)
```

You can also create sub-queries with `Query` objects like the following,

```
$userQuery = (new Query)->select('id')->from('user');
$query->where(['id' => $userQuery]);
```

which will generate the following SQL:

```
WHERE `id` IN (SELECT `id` FROM `user`)
```

Another way to use the method is the operand format which is `[operator, operand1, operand2, ...]`.

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`[10] 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.

[10] http://www.postgresql.org/docs/8.3/static/functions-matching.html#FUNCTIONS-LIKE

- `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 a `EXISTS (sub-query)` expression.
- `not exists`: similar to the `exists` operator and builds a `NOT EXISTS (sub-query)` expression.

If you are building parts of condition dynamically it's very convenient to use `andWhere()` and `orWhere()`:

```
$status = 10;
$search = 'yii';

$query->where(['status' => $status]);
if (!empty($search)) {
    $query->andWhere(['like', 'title', $search]);
}
```

In case `$search` isn't empty the following SQL will be generated:

```
WHERE (`status` = 10) AND (`title` LIKE '%yii%')
```

Building Filter Conditions When building filter conditions based on user inputs, you usually want to specially handle "empty inputs" by ignoring them in the filters. For example, you have an HTML form that takes username and email inputs. If the user only enters something in the username input, you may want to build a query that only tries to match the entered username. You may use the `filterWhere()` method achieve this goal:

```
// $username and $email are from user inputs
$query->filterWhere([
    'username' => $username,
    'email' => $email,
]);
```

The `filterWhere()` method is very similar to `where()`. The main difference is that `filterWhere()` will remove empty values from the provided condition. So if `$email` is "empty", the resulting query will be ...`WHERE username=:username`; and if both `$username` and `$email` are "empty", the query will have no `WHERE` part.

A value is *empty* if it is null, an empty string, a string consisting of whitespaces, or an empty array.

You may also use `andFilterWhere()` and `orFilterWhere()` to append more filter conditions.

`ORDER BY`

For ordering results `orderBy` and `addOrderBy` could be used:

```
$query->orderBy([
    'id' => SORT_ASC,
    'name' => SORT_DESC,
]);
```

Here we are ordering by `id` ascending and then by `name` descending.

~~~
### `GROUP BY` and `HAVING`

In order to add `GROUP BY` to generated SQL you can use the following:

```php
$query->groupBy('id, status');
~~~

If you want to add another field after using `groupBy`:

```
$query->addGroupBy(['created_at', 'updated_at']);
```

To add a `HAVING` condition the corresponding `having` method and its `andHaving` and `orHaving` can be used. Parameters for these are similar to the ones for `where` methods group:

```
$query->having(['status' => $status]);
```

`LIMIT` and `OFFSET`

To limit result to 10 rows `limit` can be used:

```
$query->limit(10);
```

To skip 100 fist rows use:

```
$query->offset(100);
```

`JOIN`

The `JOIN` clauses are generated in the Query Builder by using the applicable join method:

- `innerJoin()`
- `leftJoin()`
- `rightJoin()`

This left join selects data from two related tables in one query:

```
$query->select(['user.name AS author', 'post.title as title'])
    ->from('user')
    ->leftJoin('post', 'post.user_id = user.id');
```

In the code, the `leftJoin()` method's first parameter specifies the table to join to. The second parameter defines the join condition.

If your database application supports other join types, you can use those via the generic `join` method:

```
$query->join('FULL OUTER JOIN', 'post', 'post.user_id = user.id');
```

The first argument is the join type to perform. The second is the table to join to, and the third is the condition.

Like `FROM`, you may also join with sub-queries. To do so, specify the sub-query as an array which must contain one element. The array value must be a `Query` object representing the sub-query, while the array key is the alias for the sub-query. For example,

```
$query->leftJoin(['u' => $subQuery], 'u.id=author_id');
```

`UNION`

`UNION` in SQL adds results of one query to results of another query. Columns returned by both queries should match. In Yii in order to build it you can first form two query objects and then use `union` method:

```
$query = new Query();
$query->select("id, 'post' as type, name")->from('post')->limit(10);

$anotherQuery = new Query();
$anotherQuery->select('id, 'user' as type, name')->from('user')->limit(10);

$query->union($anotherQuery);
```

6.2.3 Batch Query

When working with large amount 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 uses of 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 since then fetched in batches in the 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 keep the memory usage under a limit.

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,

```
use yii\db\Query;

$query = (new Query())
    ->from('user')
    ->indexBy('username');

foreach ($query->batch() as $users) {
    // $users is indexed by the "username" column
}

foreach ($query->each() as $username => $user) {
}
```

6.3 Active Record

Note: This section is under development.

Active Record[11] provides an object-oriented interface for accessing data stored in a database. 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 column in that row. Instead of writing raw SQL statements, you can work with Active Record in an object-oriented fashion to manipulate the data in database tables.

For example, assume `Customer` is an Active Record class is associated with the `customer` table and `name` is a column of `customer` table. You can write the following code to insert a new row into `customer` table:

[11] http://en.wikipedia.org/wiki/Active_record_pattern

```
$customer = new Customer();
$customer->name = 'Qiang';
$customer->save();
```

The above code is equivalent to using the following raw SQL statement, which is less intuitive, more error prone, and may have compatibility problem for different DBMS:

```
$db->createCommand('INSERT INTO customer (name) VALUES (:name)', [
    ':name' => 'Qiang',
])->execute();
```

Below is the list of databases that are currently supported by Yii Active Record:

- 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 2010 or later: via `yii\db\ActiveRecord`
- Oracle: via `yii\db\ActiveRecord`
- CUBRID 9.1 or later: via `yii\db\ActiveRecord`
- Sphnix: via `yii\sphinx\ActiveRecord`, requires `yii2-sphinx` extension
- ElasticSearch: via `yii\elasticsearch\ActiveRecord`, requires `yii2-elasticsearch` extension
- Redis 2.6.12 or later: via `yii\redis\ActiveRecord`, requires `yii2-redis` extension
- MongoDB 1.3.0 or later: via `yii\mongodb\ActiveRecord`, requires `yii2-mongodb` extension

As you can see, Yii provides Active Record support for relational databases as well as NoSQL databases. 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.

6.3.1 Declaring Active Record Classes

To declare an Active Record class you need to extend `yii\db\ActiveRecord` and implement the `tableName` method that returns the name of the database table associated with the class:

```
namespace app\models;

use yii\db\ActiveRecord;

class Customer extends ActiveRecord
{
    /**
     * @return string the name of the table associated with this
    ActiveRecord class.
     */
    public static function tableName()
    {
        return 'customer';
    }
}
```

6.3.2 Accessing Column Data

Active Record maps each column of the corresponding database table row to an attribute in the Active Record object. An attribute behaves like a regular object public property. The name of an attribute is the same as the corresponding column name and is case-sensitive.

To read the value of a column, you can use the following syntax:

```
// "id" and "email" are the names of columns in the table associated with
    $customer ActiveRecord object
$id = $customer->id;
$email = $customer->email;
```

To change the value of a column, assign a new value to the associated property and save the object:

```
$customer->email = 'jane@example.com';
$customer->save();
```

6.3.3 Connecting to Database

Active Record uses a `yii\db\Connection` to exchange data with database. By default, it uses the `db` application component as the connection. As explained in Database basics, you may configure the `db` component in the application configuration file like follows,

```
return [
    'components' => [
        'db' => [
            'class' => 'yii\db\Connection',
            'dsn' => 'mysql:host=localhost;dbname=testdb',
            'username' => 'demo',
            'password' => 'demo',
        ],
    ],
];
```

If you are using multiple databases in your application and you want to use a different DB connection for your Active Record class, you may override the `yii\db\ActiveRecord::getDb()` method:

```
class Customer extends ActiveRecord
{
    // ...

    public static function getDb()
    {
        return \Yii::$app->db2;  // use "db2" application component
    }
}
```

6.3.4 Querying Data from Database

Active Record provides two entry methods for building DB queries and populating data into Active Record instances:

- `yii\db\ActiveRecord::find()`
- `yii\db\ActiveRecord::findBySql()`

Both methods return an `yii\db\ActiveQuery` instance, which extends `yii\db\Query`, and thus supports the same set of flexible and powerful DB query building methods, such as `where()`, `join()`, `orderBy()`, etc. The following examples demonstrate some of the possibilities.

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

// to return a single customer whose ID is 1:
$customer = Customer::find()
    ->where(['id' => 1])
    ->one();

// to return the number of *active* customers:
$count = Customer::find()
    ->where(['status' => Customer::STATUS_ACTIVE])
    ->count();

// to index the result by customer IDs:
$customers = Customer::find()->indexBy('id')->all();
// $customers array is indexed by customer IDs

// to retrieve customers using a raw SQL statement:
$sql = 'SELECT * FROM customer';
$customers = Customer::findBySql($sql)->all();
```

> Tip: In the code above `Customer::STATUS_ACTIVE` is a constant defined in `Customer`. It is a good practice to use meaningful constant names rather than hardcoded strings or numbers in your code.

Two shortcut methods are provided to return Active Record instances matching a primary key value or a set of column values: `findOne()` and `findAll()`. The former returns the first matching instance while the latter returns all of them. For example,

```
// to return a single customer whose ID is 1:
$customer = Customer::findOne(1);

// to return an *active* customer whose ID is 1:
$customer = Customer::findOne([
    'id' => 1,
    'status' => Customer::STATUS_ACTIVE,
]);

// to return customers whose ID is 1, 2 or 3:
$customers = Customer::findAll([1, 2, 3]);

// to return customers whose status is "deleted":
$customer = Customer::findAll([
    'status' => Customer::STATUS_DELETED,
]);
```

Retrieving Data in Arrays

Sometimes when you are processing a large amount of data, you may want to use arrays to hold the data retrieved from database to save memory. This can be done by calling `asArray()`:

```
// to return customers in terms of arrays rather than `Customer` objects:
$customers = Customer::find()
    ->asArray()
    ->all();
// each element of $customers is an array of name-value pairs
```

Retrieving Data in Batches

In Query Builder, we have explained that you may use *batch query* to keep your memory usage under a limit when querying a large amount of data from 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) {
}
```

6.3.5 Manipulating Data in Database

Active Record provides the following methods to insert, update and delete a single row in a table associated with a single Active Record instance:

- `yii\db\ActiveRecord::save()`
- `yii\db\ActiveRecord::insert()`
- `yii\db\ActiveRecord::update()`
- `yii\db\ActiveRecord::delete()`

Active Record also provides the following static methods that apply to a whole table associated with an Active Record class. Be extremely careful when using these methods as they affect the whole table. For example, `deleteAll()` will delete ALL rows in the table.

- `yii\db\ActiveRecord::updateCounters()`
- `yii\db\ActiveRecord::updateAll()`
- `yii\db\ActiveRecord::updateAllCounters()`
- `yii\db\ActiveRecord::deleteAll()`

The following examples show how to use these methods:

```
// to insert a new customer record
$customer = new Customer();
$customer->name = 'James';
$customer->email = 'james@example.com';
$customer->save();  // equivalent to $customer->insert();

// to update an existing customer record
$customer = Customer::findOne($id);
$customer->email = 'james@example.com';
$customer->save();  // equivalent to $customer->update();

// to delete an existing customer record
$customer = Customer::findOne($id);
$customer->delete();

// to delete several customers
Customer::deleteAll('age > :age AND gender = :gender', [':age' => 20, ':
    gender' => 'M']);
```

```
// to increment the age of ALL customers by 1
Customer::updateAllCounters(['age' => 1]);
```

> Info: The `save()` method will call either `insert()` or `update()`, depending on whether the Active Record instance is new or not (internally it will check the value of `yii\db\ActiveRecord::isNewRecord`). If an Active Record is instantiated via the `new` operator, calling `save()` will insert a row in the table; calling `save()` on active record fetched from database will update the corresponding row in the table.

Data Input and Validation

Because Active Record extends from `yii\base\Model`, it supports the same data input and validation features as described in Model. For example, you may declare validation rules by overwriting the `yii\base\Model::rules()` method; you may massively assign user input data to an Active Record instance; and you may call `yii\base\Model::validate()` to trigger data validation.

When you call `save()`, `insert()` or `update()`, these methods will automatically call `yii\base\Model::validate()`. If the validation fails, the corresponding data saving operation will be cancelled.

The following example shows how to use an Active Record to collect/-validate user input and save them into database:

```
// creating a new record
$model = new Customer;
if ($model->load(Yii::$app->request->post()) && $model->save()) {
    // the user input has been collected, validated and saved
}

// updating a record whose primary key is $id
$model = Customer::findOne($id);
if ($model === null) {
    throw new NotFoundHttpException;
}
if ($model->load(Yii::$app->request->post()) && $model->save()) {
    // the user input has been collected, validated and saved
}
```

Loading Default Values

Your table columns may be defined with default values. Sometimes, you may want to pre-populate your Web form for an Active Record with these values. To do so, call the `loadDefaultValues()` method before rendering the form:

```
$customer = new Customer();
$customer->loadDefaultValues();
// ... render HTML form for $customer ...
```

6.3.6 Active Record Life Cycles

It is important to understand the life cycles of Active Record when it is used to manipulate data in database. These life cycles are typically associated with corresponding events which allow you to inject code to intercept or respond to these events. They are especially useful for developing Active Record behaviors.

When instantiating a new Active Record instance, we will have the following life cycles:

1. constructor
2. `yii\db\ActiveRecord::init()`: will trigger an `yii\db\ActiveRecord::EVENT_INIT` event

When querying data through the `yii\db\ActiveRecord::find()` method, we will have the following life cycles for EVERY newly populated Active Record instance:

1. constructor
2. `yii\db\ActiveRecord::init()`: will trigger an `yii\db\ActiveRecord::EVENT_INIT` event
3. `yii\db\ActiveRecord::afterFind()`: will trigger an `yii\db\ActiveRecord::EVENT_AFTER_FIND` event

When calling `yii\db\ActiveRecord::save()` to insert or update an ActiveRecord, we will have the following life cycles:

1. `yii\db\ActiveRecord::beforeValidate()`: will trigger an `yii\db\ActiveRecord::EVENT_BEFORE_VALIDATE` event
2. `yii\db\ActiveRecord::afterValidate()`: will trigger an `yii\db\ActiveRecord::EVENT_AFTER_VALIDATE` event
3. `yii\db\ActiveRecord::beforeSave()`: will trigger an `yii\db\ActiveRecord::EVENT_BEFORE_INSERT` or `yii\db\ActiveRecord::EVENT_BEFORE_UPDATE` event
4. perform the actual data insertion or updating

5. `yii\db\ActiveRecord::afterSave()`: will trigger an `yii\db\ActiveRecord::EVENT_AFTER_INSERT` or `yii\db\ActiveRecord::EVENT_AFTER_UPDATE` event

And Finally when calling `yii\db\ActiveRecord::delete()` to delete an ActiveRecord, we will have the following life cycles:

1. `yii\db\ActiveRecord::beforeDelete()`: will trigger an `yii\db\ActiveRecord::EVENT_BEFORE_DELETE` event
2. perform the actual data deletion
3. `yii\db\ActiveRecord::afterDelete()`: will trigger an `yii\db\ActiveRecord::EVENT_AFTER_DELETE` event

6.3.7 Working with Relational Data

You can use ActiveRecord to also query a table's relational data (i.e., selection of data from Table A can also pull in related data from Table B). Thanks to ActiveRecord, the relational data returned can be accessed like a property of the ActiveRecord object associated with the primary table.

For example, with an appropriate relation declaration, by accessing `$customer->orders` you may obtain an array of `Order` objects which represent the orders placed by the specified customer.

To declare a relation, define a getter method which returns an `yii\db\ActiveQuery` object that has relation information about the relation context and thus will only query for related records. For example,

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

class Order extends \yii\db\ActiveRecord
{
    public function getCustomer()
    {
        // Order has_one Customer via Customer.id -> customer_id
        return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
    ;
    }
}
```

The methods `yii\db\ActiveRecord::hasMany()` and `yii\db\ActiveRecord::hasOne()` used in the above are used to model the many-one relationship and one-one relationship in a relational database. For example, a customer has many orders, and an order has one customer. Both methods take two parameters and return an `yii\db\ActiveQuery` object:

- `$class`: the name of the class of the related model(s). This should be a fully qualified class name.
- `$link`: the association between columns from the two tables. This should be given as an array. The keys of the array are the names of the columns from the table associated with `$class`, while the values of the array are the names of the columns from the declaring class. It is a good practice to define relationships based on table foreign keys.

After declaring relations, getting relational data is as easy as accessing a component property that is defined by the corresponding getter method:

```
// get the orders of a customer
$customer = Customer::findOne(1);
$orders = $customer->orders;  // $orders is an array of Order objects
```

Behind the scene, the above code executes the following two SQL queries, one for each line of code:

```
SELECT * FROM customer WHERE id=1;
SELECT * FROM order WHERE customer_id=1;
```

> Tip: If you access the expression `$customer->orders` again, it will not perform the second SQL query again. The SQL query is only performed the first time when this expression is accessed. Any further accesses will only return the previously fetched results that are cached internally. If you want to re-query the relational data, simply unset the existing one first: `unset($customer->orders);`.

Sometimes, you may want to pass parameters to a relational query. For example, instead of returning all orders of a customer, you may want to return only big orders whose subtotal exceeds a specified amount. To do so, declare a `bigOrders` relation with the following getter method:

```
class Customer extends \yii\db\ActiveRecord
{
    public function getBigOrders($threshold = 100)
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id'])
            ->where('subtotal > :threshold', [':threshold' => $threshold])
            ->orderBy('id');
    }
}
```

Remember that `hasMany()` returns an `yii\db\ActiveQuery` object which allows you to customize the query by calling the methods of `yii\db\ActiveQuery`.

With the above declaration, if you access `$customer->bigOrders`, it will only return the orders whose subtotal is greater than 100. To specify a different threshold value, use the following code:

```
$orders = $customer->getBigOrders(200)->all();
```

> Note: A relation method returns an instance of `yii\db\ActiveQuery`. If you access the relation like an attribute (i.e. a class property), the return value will be the query result of the relation, which could be an instance of `yii\db\ActiveRecord`, an array of that, or null, depending the multiplicity of the relation. For example, `$customer->getOrders()` returns an `ActiveQuery` instance, while `$customer->orders` returns an array of `Order` objects (or an empty array if the query results in nothing).

6.3.8 Relations with Pivot Table

Sometimes, two tables are related together via an intermediary table called pivot table[12]. To declare such relations, we can customize the `yii\db\ActiveQuery` object by calling its `yii\db\ActiveQuery::via()` or `yii\db\ActiveQuery::viaTable()` method.

For example, if table `order` and table `item` are related via pivot table `order_item`, we can declare the `items` relation in the `Order` class like the following:

```
class Order extends \yii\db\ActiveRecord
{
    public function getItems()
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->viaTable('order_item', ['order_id' => 'id']);
    }
}
```

The `yii\db\ActiveQuery::via()` method is similar to `yii\db\ActiveQuery::viaTable()` except that the first parameter of `yii\db\ActiveQuery::via()` takes a relation name declared in the ActiveRecord class instead of the pivot table name. For example, the above `items` relation can be equivalently declared as follows:

```
class Order extends \yii\db\ActiveRecord
{
    public function getOrderItems()
    {
        return $this->hasMany(OrderItem::className(), ['order_id' => 'id']);
    }

    public function getItems()
```

[12]Pivot table on Wikipedia: `http://en.wikipedia.org/wiki/Pivot_table`

```
    {
        return $this->hasMany(Item::className(), ['id' => 'item_id'])
            ->via('orderItems');
    }
}
```

6.3.9 Lazy and Eager Loading

As described earlier, when you access the related objects the first time, ActiveRecord will perform a DB query to retrieve the corresponding data and populate it into the related objects. No query will be performed if you access the same related objects again. We call this *lazy loading*. For example,

```
// SQL executed: SELECT * FROM customer WHERE id=1
$customer = Customer::findOne(1);
// SQL executed: SELECT * FROM order WHERE customer_id=1
$orders = $customer->orders;
// no SQL executed
$orders2 = $customer->orders;
```

Lazy loading is very convenient to use. However, it may suffer from a performance issue in the following scenario:

```
// SQL executed: SELECT * FROM customer LIMIT 100
$customers = Customer::find()->limit(100)->all();

foreach ($customers as $customer) {
    // SQL executed: SELECT * FROM order WHERE customer_id=...
    $orders = $customer->orders;
    // ...handle $orders...
}
```

How many SQL queries will be performed in the above code, assuming there are more than 100 customers in the database? 101! The first SQL query brings back 100 customers. Then for each customer, a SQL query is performed to bring back the orders of that customer.

To solve the above performance problem, you can use the so-called *eager loading* approach by calling `yii\db\ActiveQuery::with()`:

```
// SQL executed: SELECT * FROM customer LIMIT 100;
//               SELECT * FROM orders WHERE customer_id IN (1,2,...)
$customers = Customer::find()->limit(100)
    ->with('orders')->all();

foreach ($customers as $customer) {
    // no SQL executed
    $orders = $customer->orders;
    // ...handle $orders...
}
```

As you can see, only two SQL queries are needed for the same task!

> Info: In general, if you are eager loading `N` relations among which `M` relations are defined with `via()` or `viaTable()`, a total number of `1+M+N` SQL queries will be performed: one query to bring back the rows for the primary table, one for each of the `M` pivot tables corresponding to the `via()` or `viaTable()` calls, and one for each of the `N` related tables.

> Note: When you are customizing `select()` with eager loading, make sure you include the columns that link the related models. Otherwise, the related models will not be loaded. 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 following:
$orders = Order::find()->select(['id', 'amount', 'customer_id'])->with('
    customer')->all();
```

Sometimes, you may want to customize the relational queries on the fly. This can be done for both lazy loading and eager loading. For example,

```
$customer = Customer::findOne(1);
// lazy loading: SELECT * FROM order WHERE customer_id=1 AND subtotal>100
$orders = $customer->getOrders()->where('subtotal>100')->all();

// eager loading: SELECT * FROM customer LIMIT 100
//                SELECT * FROM order WHERE customer_id IN (1,2,...) AND
    subtotal>100
$customers = Customer::find()->limit(100)->with([
    'orders' => function($query) {
        $query->andWhere('subtotal>100');
    },
])->all();
```

6.3.10 Inverse Relations

Relations can often be defined in pairs. For example, `Customer` may have a relation named `orders` while `Order` may have a relation named `customer`:

```
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'])
    ;
    }
}
```

If we perform the following query, we would find that the `customer` of an order is not the same customer object that finds those orders, and accessing `customer->orders` will trigger one SQL execution while accessing the `customer` of an order will trigger another SQL execution:

```
// SELECT * FROM customer WHERE id=1
$customer = Customer::findOne(1);
// echoes "not equal"
// SELECT * FROM order WHERE customer_id=1
// SELECT * FROM customer WHERE id=1
if ($customer->orders[0]->customer === $customer) {
    echo 'equal';
} else {
    echo 'not equal';
}
```

To avoid the redundant execution of the last SQL statement, we could declare the inverse relations for the `customer` and the `orders` relations by calling the `yii\db\ActiveQuery::inverseOf()` method, like the following:

```
class Customer extends ActiveRecord
{
    ....
    public function getOrders()
    {
        return $this->hasMany(Order::className(), ['customer_id' => 'id'])->
    inverseOf('customer');
    }
}
```

Now if we execute the same query as shown above, we would get:

```
// SELECT * FROM customer WHERE id=1
$customer = Customer::findOne(1);
// echoes "equal"
// SELECT * FROM order WHERE customer_id=1
if ($customer->orders[0]->customer === $customer) {
    echo 'equal';
} else {
    echo 'not equal';
}
```

In the above, we have shown how to use inverse relations in lazy loading. Inverse relations also apply in eager loading:

```
// SELECT * FROM customer
// SELECT * FROM order WHERE customer_id IN (1, 2, ...)
$customers = Customer::find()->with('orders')->all();
// echoes "equal"
```

```
if ($customers[0]->orders[0]->customer === $customers[0]) {
    echo 'equal';
} else {
    echo 'not equal';
}
```

> Note: Inverse relation cannot be defined with a relation that involves pivoting tables. That is, if your relation is defined with `yii\db\ActiveQuery::via()` or `yii\db\ActiveQuery::viaTable()`, you cannot call `yii\db\ActiveQuery::inverseOf()` further.

6.3.11 Joining with Relations

When working with relational databases, a common task is to join multiple tables and apply various query conditions and parameters to the JOIN SQL statement. Instead of calling `yii\db\ActiveQuery::join()` explicitly to build up the JOIN query, you may reuse the existing relation definitions and call `yii\db\ActiveQuery::joinWith()` to achieve this goal. For example,

```
// find all orders and sort the orders by the customer id and the order id.
    also eager loading "customer"
$orders = Order::find()->joinWith('customer')->orderBy('customer.id, order.
    id')->all();
// find all orders that contain books, and eager loading "books"
$orders = Order::find()->innerJoinWith('books')->all();
```

In the above, the method `yii\db\ActiveQuery::innerJoinWith()` is a shortcut to `yii\db\ActiveQuery::joinWith()` with the join type set as `INNER JOIN`.

You may join with one or multiple relations; you may apply query conditions to the relations on-the-fly; and you may also join with sub-relations. For example,

```
// join with multiple relations
// find out the orders that contain books and are placed by customers who
    registered within the past 24 hours
$orders = Order::find()->innerJoinWith([
    'books',
    'customer' => function ($query) {
        $query->where('customer.created_at > ' . (time() - 24 * 3600));
    }
])->all();
// join with sub-relations: join with books and books' authors
$orders = Order::find()->joinWith('books.author')->all();
```

Behind the scene, Yii will first execute a JOIN SQL statement to bring back the primary models satisfying the conditions applied to the JOIN SQL. It will then execute a query for each relation and populate the corresponding related records.

The difference between `yii\db\ActiveQuery::joinWith()` and `yii\db\ActiveQuery::with()` is that the former joins the tables for the primary model class and the related model classes to retrieve the primary models, while the latter just queries against the table for the primary model class to retrieve the primary models.

Because of this difference, you may apply query conditions that are only available to a JOIN SQL statement. For example, you may filter the primary models by the conditions on the related models, like the example above. You may also sort the primary models using columns from the related tables.

When using `yii\db\ActiveQuery::joinWith()`, you are responsible to disambiguate column names. In the above examples, we use `item.id` and `order.id` to disambiguate the `id` column references because both of the order table and the item table contain a column named `id`.

By default, when you join with a relation, the relation will also be eagerly loaded. You may change this behavior by passing the `$eagerLoading` parameter which specifies whether to eager load the specified relations.

And also by default, `yii\db\ActiveQuery::joinWith()` uses `LEFT JOIN` to join the related tables. You may pass it with the `$joinType` parameter to customize the join type. As a shortcut to the `INNER JOIN` type, you may use `yii\db\ActiveQuery::innerJoinWith()`.

Below are some more examples,

```
// find all orders that contain books, but do not eager loading "books".
$orders = Order::find()->innerJoinWith('books', false)->all();
// which is equivalent to the above
$orders = Order::find()->joinWith('books', false, 'INNER JOIN')->all();
```

Sometimes when joining two tables, you may need to specify some extra condition in the ON part of the JOIN query. This can be done by calling the `yii\db\ActiveQuery::onCondition()` method like the following:

```
class User extends ActiveRecord
{
    public function getBooks()
    {
        return $this->hasMany(Item::className(), ['owner_id' => 'id'])->
    onCondition(['category_id' => 1]);
    }
}
```

In the above, the `yii\db\ActiveRecord::hasMany()` method returns an `yii\db\ActiveQuery` instance, upon which `yii\db\ActiveQuery::onCondition()` is called to specify that only items whose `category_id` is 1 should be returned.

When you perform query using `yii\db\ActiveQuery::joinWith()`, the on-condition will be put in the ON part of the corresponding JOIN query. For example,

```
// SELECT user.* FROM user LEFT JOIN item ON item.owner_id=user.id AND
    category_id=1
// SELECT * FROM item WHERE owner_id IN (...) AND category_id=1
$users = User::find()->joinWith('books')->all();
```

Note that if you use eager loading via `yii\db\ActiveQuery::with()` or lazy loading, the on-condition will be put in the WHERE part of the corresponding SQL statement, because there is no JOIN query involved. For example,

```
// SELECT * FROM user WHERE id=10
$user = User::findOne(10);
// SELECT * FROM item WHERE owner_id=10 AND category_id=1
$books = $user->books;
```

6.3.12 Working with Relationships

ActiveRecord provides the following two methods for establishing and breaking a relationship between two ActiveRecord objects:

- `yii\db\ActiveRecord::link()`
- `yii\db\ActiveRecord::unlink()`

For example, given a customer and a new order, we can use the following code to make the order owned by the customer:

```
$customer = Customer::findOne(1);
$order = new Order();
$order->subtotal = 100;
$customer->link('orders', $order);
```

The `yii\db\ActiveRecord::link()` call above will set the `customer_id` of the order to be the primary key value of `$customer` and then call `yii\db\ActiveRecord::save()` to save the order into database.

6.3.13 Cross-DBMS Relations

ActiveRecord allows to establish relationship between entities from different DBMS. For example: between relational database table and MongoDB collection. Such relation does not require any special code:

```
// Relational database Active Record
class Customer extends \yii\db\ActiveRecord
{
    public static function tableName()
    {
        return 'customer';
    }

    public function getComments()
    {
        // Customer, stored in relational database, has many Comments,
    stored in MongoDB collection:
        return $this->hasMany(Comment::className(), ['customer_id' => 'id'])
    ;
    }
}

// MongoDb Active Record
class Comment extends \yii\mongodb\ActiveRecord
{
    public static function collectionName()
    {
        return 'comment';
    }

    public function getCustomer()
    {
        // Comment, stored in MongoDB collection, has one Customer, stored
     in relational database:
        return $this->hasOne(Customer::className(), ['id' => 'customer_id'])
    ;
    }
}
```

All Active Record features like eager and lazy loading, establishing and breaking a relationship and so on, are available for cross-DBMS relations.

> Note: do not forget Active Record solutions for different DBMS may have specific methods and features, which may not be applied for cross-DBMS relations. For example: usage of `yii\db\ActiveQuery::joinWith()` will obviously not work with relation to the MongoDB collection.

6.3.14 Scopes

When you call `yii\db\ActiveRecord::find()` or `yii\db\ActiveRecord::findBySql()`, it returns an `yii\db\ActiveQuery` instance. You may call additional query methods, such as `yii\db\ActiveQuery::where()`, `yii\db\ActiveQuery::orderBy()`, to further specify the query conditions.

It is possible that you may want to call the same set of query methods in different places. If this is the case, you should consider defining the so-called *scopes*. A scope is essentially a method defined in a custom query class that calls a set of query methods to modify the query object. You can then use a scope like calling a normal query method.

Two steps are required to define a scope. First create a custom query class for your model and define the needed scope methods in this class. For example, create a `CommentQuery` class for the `Comment` model and define the `active()` scope method like the following:

```
namespace app\models;

use yii\db\ActiveQuery;

class CommentQuery extends ActiveQuery
{
    public function active($state = true)
    {
        $this->andWhere(['active' => $state]);
        return $this;
    }
}
```

Important points are:

1. Class should extend from `yii\db\ActiveQuery` (or another `ActiveQuery` such as `yii\mongodb\ActiveQuery`).
2. A method should be `public` and should return `$this` in order to allow method chaining. It may accept parameters.
3. Check `yii\db\ActiveQuery` methods that are very useful for modifying query conditions.

Second, override `yii\db\ActiveRecord::find()` to use the custom query class instead of the regular `yii\db\ActiveQuery`. For the example above, you need to write the following code:

```
namespace app\models;

use yii\db\ActiveRecord;

class Comment extends ActiveRecord
{
    /**
     * @inheritdoc
     * @return CommentQuery
     */
    public static function find()
    {
        return new CommentQuery(get_called_class());
    }
}
```

That's it. Now you can use your custom scope methods:

```
$comments = Comment::find()->active()->all();
$inactiveComments = Comment::find()->active(false)->all();
```

You can also use scopes when defining relations. For example,

```
class Post extends \yii\db\ActiveRecord
{
    public function getActiveComments()
    {
        return $this->hasMany(Comment::className(), ['post_id' => 'id'])->
    active();

    }
}
```

Or use the scopes on-the-fly when performing relational query:

```
$posts = Post::find()->with([
    'comments' => function($q) {
        $q->active();
    }
])->all();
```

Default Scope

If you used Yii 1.1 before, you may know a concept called *default scope*. A default scope is a scope that applies to ALL queries. You can define a default scope easily by overriding `yii\db\ActiveRecord::find()`. For example,

```
public static function find()
{
    return parent::find()->where(['deleted' => false]);
}
```

Note that all your queries should then not use `yii\db\ActiveQuery::where()` but `yii\db\ActiveQuery::andWhere()` and `yii\db\ActiveQuery::orWhere()` to not override the default condition.

6.3.15 Transactional operations

When a few DB operations are related and are executed

TODO: FIXME: WIP, TBD, `https://github.com/yiisoft/yii2/issues/226`

, `yii\db\ActiveRecord::afterSave()`, `yii\db\ActiveRecord::beforeDelete()` and/or `yii\db\ActiveRecord::afterDelete()` life cycle methods. Developer may come to the solution of overriding ActiveRecord `yii\db\ActiveRecord::save()` method with database transaction wrapping or even using transaction in controller action, which is strictly speaking doesn't seem to be a good practice (recall "skinny-controller / fat-model" fundamental rule).

Here these ways are (**DO NOT** use them unless you're sure what you are actually doing). Models:

```
class Feature extends \yii\db\ActiveRecord
{
    // ...

    public function getProduct()
    {
        return $this->hasOne(Product::className(), ['id' => 'product_id']);
    }
}

class Product extends \yii\db\ActiveRecord
{
    // ...
    public function getFeatures()
    {
        return $this->hasMany(Feature::className(), ['product_id' => 'id']);
    }
}
```

Overriding `yii\db\ActiveRecord::save()` method:

```

class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

Using transactions within controller layer:

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

Instead of using these fragile methods you should consider using atomic scenarios and operations feature.

```
class Feature extends \yii\db\ActiveRecord
{
    // ...

    public function getProduct()
    {
        return $this->hasOne(Product::className(), ['product_id' => 'id']);
    }

    public function scenarios()
    {
        return [
            'userCreates' => [
                'attributes' => ['name', 'value'],
                'atomic' => [self::OP_INSERT],
            ],
        ];
    }
}

class Product extends \yii\db\ActiveRecord
{
    // ...

    public function getFeatures()
    {
        return $this->hasMany(Feature::className(), ['id' => 'product_id']);
    }

    public function scenarios()
    {
        return [
            'userCreates' => [
                'attributes' => ['title', 'price'],
                'atomic' => [self::OP_INSERT],
            ],
        ];
    }

    public function afterValidate()
    {
        parent::afterValidate();
        // FIXME: TODO: WIP, TBD
    }

    public function afterSave($insert)
    {
        parent::afterSave($insert);
        if ($this->getScenario() === 'userCreates') {
            // FIXME: TODO: WIP, TBD
        }
    }
}
```

Controller is very thin and neat:

```
class ProductController extends \yii\web\Controller
{
    public function actionCreate()
    {
        // FIXME: TODO: WIP, TBD
    }
}
```

6.3.16 Optimistic Locks

TODO

6.3.17 Dirty Attributes

TODO

6.3.18 See also

- Model
- `yii\db\ActiveRecord`

6.4 Database Migration

> Note: This section is under development.

Like source code, the structure of a database evolves as a database-driven application is developed and maintained. For example, during development, a new table may be added; Or, after the application goes live, it may be discovered that an additional index is required. It is important to keep track of these structural database changes (called **migration**), just as changes to the source code is tracked using version control. If the source code and the database become out of sync, bugs will occur, or the whole application might break. For this reason, Yii provides a database migration tool that can keep track of database migration history, apply new migrations, or revert existing ones.

The following steps show how database migration is 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 syncing his database to reflect the changes Tim made.

Yii supports database migration via the `yii migrate` command line tool. This tool supports:

- Creating new migrations
- Applying, reverting, and redoing migrations
- Showing migration history and new migrations

6.4.1 Creating Migrations

To create a new migration, run the following command:

```
yii migrate/create <name>
```

The required `name` parameter specifies a very brief description of the migration. For example, if the migration creates a new table named *news*, you'd use the command:

```
yii migrate/create create_news_table
```

As you'll shortly see, the `name` parameter is used as part of a PHP class name in the migration. Therefore, it should only contain letters, digits and/or underscore characters.

The above command will create a new file named `m101129_185401_create_news_table.php`. This file will be created within the `@app/migrations` directory. Initially, the migration file will be generated with the following code:

```
class m101129_185401_create_news_table extends \yii\db\Migration
{
    public function up()
    {
    }

    public function down()
    {
        echo "m101129_185401_create_news_table cannot be reverted.\n";
        return false;
    }
}
```

Notice that the class name is the same as the file name, and follows the pattern `m<timestamp>_<name>`, where:

- `<timestamp>` refers to the UTC timestamp (in the format of `yymmdd_hhmmss`) when the migration is created,
- `<name>` is taken from the command's `name` parameter.

In the class, the `up()` method should contain the code implementing the actual database migration. In other words, the `up()` method executes code that actually changes the database. The `down()` method may contain code that reverts the changes made by `up()`.

Sometimes, it is impossible for the `down()` to undo the database migration. For example, if the migration deletes table rows or an entire table, that data cannot be recovered in the `down()` method. In such cases, the migration is called irreversible, meaning the database cannot be rolled back to a previous state. When a migration is irreversible, as in the above generated code, the `down()` method returns `false` to indicate that the migration cannot be reverted.

As an example, let's show the migration about creating a news table.

```

use yii\db\Schema;

class m101129_185401_create_news_table extends \yii\db\Migration
{
    public function up()
    {
        $this->createTable('news', [
            'id' => 'pk',
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
    }

    public function down()
    {
        $this->dropTable('news');
    }

}
```

The base class [\yii\db\Migration] exposes a database connection via `db` property. You can use it for manipulating data and schema of a database.

The column types used in this example are abstract types that will be replaced by Yii with the corresponding types depended on your database management system. You can use them to write database independent migrations. For example `pk` will be replaced by `int(11) NOT NULL AUTO_INCREMENT PRIMARY KEY` for MySQL and `integer PRIMARY KEY AUTOINCREMENT NOT NULL` for sqlite. See documentation of `yii\db\QueryBuilder::getColumnType()` for more details and a list of available types. You may also use the constants defined in `yii\db\Schema` to define column types.

> Note: You can add constraints and other custom table options at the end of the table description by specifying them as simple string. For example in the above migration, after `content` attribute definition you can write `'CONSTRAINT ...'` or other custom options.

6.4.2 Transactional Migrations

While performing complex DB migrations, we usually want to make sure that each migration succeed or fail as a whole so that the database maintains the consistency and integrity. In order to achieve this goal, we can exploit DB transactions. We could use special methods `safeUp` and `safeDown` for these purposes.

```
use yii\db\Schema;

class m101129_185401_create_news_table extends \yii\db\Migration
{
    public function safeUp()
    {
        $this->createTable('news', [
            'id' => 'pk',
            'title' => Schema::TYPE_STRING . ' NOT NULL',
            'content' => Schema::TYPE_TEXT,
        ]);
        $this->createTable('user', [
            'id' => 'pk',
            'login' => Schema::TYPE_STRING . ' NOT NULL',
            'password' => Schema::TYPE_STRING . ' NOT NULL',
        ]);
    }

    public function safeDown()
    {
        $this->dropTable('news');
        $this->dropTable('user');
    }

}
```

When your code uses more then one query it is recommended to use `safeUp` and `safeDown`.

> Note: Not all DBMS support transactions. And some DB queries cannot be put into a transaction. In this case, you will have to implement `up()` and `down()`, instead. And for MySQL, some SQL statements may cause implicit commit[13].

6.4.3 Applying Migrations

To apply all available new migrations (i.e., make the local database up-to-date), run the following command:

```
yii migrate
```

The command will show the list of all new migrations. If you confirm to apply the migrations, it will run the `up()` method in every new migration class, one after another, in the order of the timestamp value in the class name.

After applying a migration, the migration tool will keep a record in a database table named `migration`. This allows the tool to identify which migrations have been applied and which are not. If the `migration` table does not exist, the tool will automatically create it in the database specified by the `db` application component.

Sometimes, we may only want to apply one or a few new migrations. We can use the following command:

```
yii migrate/up 3
```

This command will apply the 3 new migrations. Changing the value 3 will allow us to change the number of migrations to be applied.

We can also migrate the database to a specific version with the following command:

[13] `http://dev.mysql.com/doc/refman/5.1/en/implicit-commit.html`

```
yii migrate/to 101129_185401
```

That is, we use the timestamp part of a migration name to specify the version that we want to migrate the database to. If there are multiple migrations between the last applied migration and the specified migration, all these migrations will be applied. If the specified migration has been applied before, then all migrations applied after it will be reverted (to be described in the next section).

6.4.4 Reverting Migrations

To revert the last one or several applied migrations, we can use the following command:

```
yii migrate/down [step]
```

where the optional `step` parameter specifies how many migrations to be reverted back. It defaults to 1, meaning reverting back the last applied migration.

As we described before, not all migrations can be reverted. Trying to revert such migrations will throw an exception and stop the whole reverting process.

6.4.5 Redoing Migrations

Redoing migrations means first reverting and then applying the specified migrations. This can be done with the following command:

```
yii migrate/redo [step]
```

where the optional `step` parameter specifies how many migrations to be redone. It defaults to 1, meaning redoing the last migration.

6.4.6 Showing Migration Information

Besides applying and reverting migrations, the migration tool can also display the migration history and the new migrations to be applied.

```
yii migrate/history [limit]
yii migrate/new [limit]
```

where the optional parameter `limit` specifies the number of migrations to be displayed. If `limit` is not specified, all available migrations will be displayed.

The first command shows the migrations that have been applied, while the second command shows the migrations that have not been applied.

6.4.7 Modifying Migration History

Sometimes, we may want to modify the migration history to a specific migration version without actually applying or reverting the relevant migrations.

This often happens when developing a new migration. We can use the following command to achieve this goal.

```
yii migrate/mark 101129_185401
```

This command is very similar to `yii migrate/to` command, except that it only modifies the migration history table to the specified version without applying or reverting the migrations.

6.4.8 Customizing Migration Command

There are several ways to customize the migration command.

Use Command Line Options

The migration command comes with four options that can be specified in command line:

- `interactive`: boolean, specifies whether to perform migrations in an interactive mode. Defaults to true, meaning the user will be prompted when performing a specific migration. You may set this to false should the migrations be done in a background process.
- `migrationPath`: string, specifies the directory storing all migration class files. This must be specified in terms of a path alias, and the corresponding directory must exist. If not specified, it will use the `migrations` sub-directory under the application base path.
- `migrationTable`: string, specifies the name of the database table for storing migration history information. It defaults to `migration`. The table structure is `version varchar(255) primary key, apply_time integer`.
- `connectionID`: string, specifies the ID of the database application component. Defaults to 'db'.
- `templateFile`: string, specifies the path of the file to be served as the code template for generating the migration classes. This must be specified in terms of a path alias (e.g. `application.migrations.template`). If not set, an internal template will be used. Inside the template, the token `{ClassName}` will be replaced with the actual migration class name.

To specify these options, execute the migrate command using the following format

```
yii migrate/up --option1=value1 --option2=value2 ...
```

For example, if we want to migrate for a `forum` module whose migration files are located within the module's `migrations` directory, we can use the following command:

```
yii migrate/up --migrationPath=@app/modules/forum/migrations
```

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, we may want to use a different table to store the migration history, or we may want to use a customized migration template. We can do so by modifying the console application's configuration file like the following,

```
'controllerMap' => [
    'migrate' => [
        'class' => 'yii\console\controllers\MigrateController',
        'migrationTable' => 'my_custom_migrate_table',
    ],
]
```

Now if we run the `migrate` command, the above configurations will take effect without requiring us to enter the command line options every time. Other command options can be also configured this way.

Error: not existing file: db-sphinx.md

Error: not existing file: db-redis.md

Error: not existing file: db-mongodb.md

Error: not existing file: db-elastic-search.md

Chapter 7

Getting Data from Users

7.1 Working with Forms

> Note: This section is under development.

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.

When creating model-based forms, the first step is to define the model itself. The model can be either based upon the Active Record class, or the more generic Model class. For this login example, a generic model will be used:

```
use yii\base\Model;

class LoginForm extends Model
{
    public $username;
    public $password;

    /**
     * @return array the validation rules.
     */
    public function rules()
    {
        return [
            // username and password are both required
            [['username', 'password'], 'required'],
            // password is validated by validatePassword()
            ['password', 'validatePassword'],
        ];
    }

    /**
     * Validates the password.
     * This method serves as the inline validation for password.
     */
    public function validatePassword()
    {
        $user = User::findByUsername($this->username);
        if (!$user || !$user->validatePassword($this->password)) {
            $this->addError('password', 'Incorrect username or password.');
        }
    }

    /**
     * Logs in a user using the provided username and password.
     * @return boolean whether the user is logged in successfully
     */
    public function login()
    {
        if ($this->validate()) {
            $user = User::findByUsername($this->username);
            return true;
        } else {
            return false;
        }
    }
}
```

The controller will pass an instance of that model to the view, wherein the `yii\widgets\ActiveForm` widget is used:

```
use yii\helpers\Html;
use yii\widgets\ActiveForm;

<?php $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() ?>
```

In the above code, `yii\widgets\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()` and `yii\widgets\ActiveForm::end()` will be wrapped within the `<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.

In order to create a form element in the form, along with the element's label, and any application JavaScript validation, the `yii\widgets\ActiveForm::field()` method of the Active Form widget is called. When the invocation of this method is echoed directly, the result is a regular (text) input. To customize the output, you can chain additional methods to this call:

```
<?= $form->field($model, 'password')->passwordInput() ?>

// or

<?= $form->field($model, 'username')->textInput()->hint('Please enter your
    name')->label('Name') ?>
```

This will create all the `<label>`, `<input>` and other tags according to the template defined by the form field. To add these tags yourself you can use the `Html` helper class.

If you want to use one of HTML5 fields you may specify input type directly like the following:

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

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']);
```

Tip: in order to style required fields with asterisk you can use the following CSS:

```
div.required label:after {
    content: " *";
    color: red;
}
```

7.1.1 Handling multiple models with a single form

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 name-value and is represented by `Setting` model. The following shows how to implement it with Yii.

Let's start with controller action:

```
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::validateMultiple($settings)) {
            foreach ($settings as $setting) {
                $setting->save(false);
            }

            return $this->redirect('index');
        }

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

In the code above we're using `indexBy` when retrieving models from database to make array indexed by model ids. These will be later used to identify form fields. `loadMultiple` fills multiple modelds with the form data coming from POST and `validateMultiple` validates all models at once. In order to skip validation when saving we're passing `false` as a parameter to `save`.

Now the form that's in `update` view:

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

$form = ActiveForm::begin();

foreach ($settings as $index => $setting) {
    echo Html::encode($setting->name) . ': ' . $form->field($setting, "[
    $index]value");
}

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 `loadMultiple` determines which model to fill with which values.

7.2 Validating Input

As a rule of thumb, you should never trust the data received from end users and should always validate them before putting them 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 succeeds 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
$model->attributes = \Yii::$app->request->post('ContactForm');

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

Behind the scene, the `validate()` method 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`. 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`. These rules are called *active rules*.

3. Use each active rule to validate each active attribute associated with that rule. If the rule fails, keep an error message for the attribute in the model.

7.2.1 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()` 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 instead of 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
    '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.
- the name of a validator class. 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 checking the current `yii\base\Model::scenario` against the scenarios declared in `yii\base\Model::scenarios()`. These attributes are the active attributes.

2. Determine which rules should be applied by checking the current yii\base\Model::scenario against the rules declared in yii\base\Model::rules(). These rules are the active rules.

3. Use each active rule to validate each active attribute which is associated with the rule.

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()`.

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 validator will add a message "Username cannot be blank." to a model when its `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 validator supports yii\validators\NumberValidator::tooBig and yii\validators\NumberValidator::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.

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 property to define such conditions. For example,

```
[
    ['state', 'required', 'when' => function($model) {
        return $model->country == 'USA';
    }],
]
```

The `yii\validators\Validator::when` property takes a PHP callable with the following signature:

```
/**
 * @param Model $model the model being validated
 * @param string $attribute the attribute being validated
 * @return boolean whether the rule should be applied
 */
function ($model, $attribute)
```

If you also need to support client-side conditional validation, you should configure the `yii\validators\Validator::whenClient` 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').value == '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 rule declaration shows how to trim the spaces in the input by using the trim core validator:

```
[
    ['username', 'trim'],
]
```

You may also use the more general filter validator if your data filtering need is more complex than space trimming.

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.

7.2.2 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()` 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 such kind 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 succeeds or not by calling the `yii\base\DynamicModel::hasErrors()` method, and then get the validation errors from the `yii\base\DynamicModel::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`.

7.2.3 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 array $params the additional name-value pairs given in the rule
 */
function ($model, $attribute)
```

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, $params) {
                if (!ctype_alnum($this->$attribute)) {
                    $this->addError($attribute, 'The token must contain
    letters or digits.');
                }
            }],
        ];
    }

    public function validateType($attribute, $params)
    {
        if (!in_array($this->$attribute, ['USA', 'Web'])) {
            $this->addError($attribute, 'The country must be either "USA" or
     "Web".');
        }
    }
}
```

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,

```
namespace app\components;

use yii\validators\Validator;

class CountryValidator extends Validator
{
    public function validateAttribute($model, $attribute)
    {
        if (!in_array($model->$attribute, ['USA', 'Web'])) {
            $this->addError($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()`.

Handling Empty Inputs

Validators often need to check if an input is empty or not. You may call `yii\validators\Validator::isEmpty()` to perform this check. By default, this method will return true if a value is an empty string, an empty array or null.

Users of validators can customize the default empty detection logic by configuring the `yii\validators\Validator::isEmpty` property. For example,

```
[
    ['agree', 'required', 'isEmpty' => function ($value) {
        return empty($value);
    }],
]
```

7.2.4 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 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. It main purpose is to provider users better experience. Like 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()`, like described in the previous subsections.

Using Client-Side Validation

Many core validators support client-side validation out-of-box. All you need to do is just to 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 do not want client-side validation, you may simply configure the `yii\widgets\ActiveForm::enableClientValidation` property to be false.

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.

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);
        return <<<JS
if (!$.inArray(value, $statuses)) {
    messages.push($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: ‘php [
>
> ```
> ['status', 'in', 'range' => Status::find()->select('id')->
>     asArray()->column()],
> ```
>
>] ‘

7.3 Uploading Files

> Note: This section is under development.

Uploading files in Yii is done via form model, its validation rules and some controller code. Let's review what's needed to handle uploads properly.

7.3.1 Form model

First of all, you need to create a model that will handle file upload. Create `models/UploadForm.php` with the following content:

```
namespace app\models;

use yii\base\Model;
use yii\web\UploadedFile;

/**
 * UploadForm is the model behind the upload form.
 */
class UploadForm extends Model
{
    /**
     * @var UploadedFile|Null file attribute
     */
    public $file;

    /**
     * @return array the validation rules.
     */
    public function rules()
    {
        return [
            [['file'], 'file'],
        ];
    }
}
```

In the code above, we created a model `UploadForm` with an attribute `$file` that will become `<input type="file">` in the HTML form. The attribute has the validation rule named `file` that uses `yii\validators\FileValidator`.

7.3.2 Form view

Next create a view that will render the form.

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

$form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-data'
    ]]); ?>

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

<button>Submit</button>

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

The `'enctype' => 'multipart/form-data'` is important since it allows file uploads. `fileInput()` represents a form input field.

7.3.3 Controller

Now create the controller that connects form and model together:

```
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) {
            $model->file = UploadedFile::getInstance($model, 'file');

            if ($model->validate()) {
                $model->file->saveAs('uploads/' . $model->file->baseName . '
    .' . $model->file->extension);
            }
        }

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

Instead of `model->load(...)` we are using `UploadedFile::getInstance(...)`. `\yii\web\UploadedFile` does not run the model validation. It only provides information about the uploaded file. Therefore, you need to run validation manually via `$model->validate()`. This triggers the `yii\validators\FileValidator` that expects a file:

```
$file instanceof UploadedFile || $file->error == UPLOAD_ERR_NO_FILE //in
    code framework
```

If validation is successful, then we're saving the file:

```
$model->file->saveAs('uploads/' . $model->file->baseName . '.' . $model->
    file->extension);
```

If you're using "basic" application template then folder `uploads` should be created under `web`.

That's it. Load the page and try uploading. Uplaods should end up in `basic/web/uploads`.

7.3.4 Additional information

Required rule

If you need to make file upload mandatory use `skipOnEmpty` like the following:

```
public function rules()
{
    return [
        [['file'], 'file', 'skipOnEmpty' => false],
    ];
}
```

MIME type

It is wise to validate type of the file uploaded. FileValidator has property `$types` for the purpose:

```
public function rules()
{
    return [
        [['file'], 'file', 'types' => 'gif, jpg',],
    ];
}
```

The thing is that it validates only file extension and not the file content. In order to validate content as well use `mimeTypes` property of `ImageValidator`:

```
public function rules()
{
    return [
        [['file'], 'image', 'mimeTypes' => 'image/jpeg, image/png',],
    ];
}
```

Uploading multiple files

If you need download multiple files at once some adjustments are required. View:

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

$form = ActiveForm::begin(['options' => ['enctype' => 'multipart/form-data'
    ]]);

if ($model->hasErrors()) { //it is necessary to see all the errors for all
    the files.
    echo '<pre>';
    print_r($model->getErrors());
    echo '</pre>';
}
?>

<?= $form->field($model, 'file[]')->fileInput(['multiple' => '']) ?>

    <button>Submit</button>

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

The difference is the following line:

```
<?= $form->field($model, 'file[]')->fileInput(['multiple' => '']) ?>
```

Controller:

```
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) {

            $files = UploadedFile::getInstances($model, 'file');

            foreach ($files as $file) {

                $_model = new UploadForm();

                $_model->file = $file;

                if ($_model->validate()) {
                    $_model->file->saveAs('uploads/' . $_model->file->
    baseName . '.' . $_model->file->extension);
                } else {
                    foreach ($_model->getErrors('file') as $error) {
                        $model->addError('file', $error);
                    }
                }
            }

            if ($model->hasErrors('file')){
                $model->addError(
                    'file',
                    count($model->getErrors('file')) . ' of ' . count($files
    ) . ' files not uploaded'
                );
            }

        }

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

The difference is `UploadedFile::getInstances($model, 'file');` instead of `UploadedFile::getInstance($model, 'file');`. Former returns instances for **all** uploaded files while the latter gives you only a single instance.

Error: not existing file: input-multiple-models.md

Chapter 8

Displaying Data

Error: not existing file: output-formatting.md

Error: not existing file: output-pagination.md

Error: not existing file: output-sorting.md

8.1 Data providers

> Note: This section is under development.

Data provider abstracts data set via `yii\data\DataProviderInterface` and handles pagination and sorting. It can be used by grids, lists and other data widgets.

In Yii there are three built-in data providers: `yii\data\ActiveDataProvider`, `yii\data\ArrayDataProvider` and `yii\data\SqlDataProvider`.

8.1.1 Active data provider

`ActiveDataProvider` provides data by performing DB queries using `yii\db\Query` and `yii\db\ActiveQuery`.

The following is an example of using it to provide ActiveRecord instances:

```
$provider = new ActiveDataProvider([
    'query' => Post::find(),
    'pagination' => [
        'pageSize' => 20,
    ],
]);

// get the posts in the current page
$posts = $provider->getModels();
```

And the following example shows how to use ActiveDataProvider without ActiveRecord:

```
$query = new Query();
$provider = new ActiveDataProvider([
    'query' => $query->from('post'),
    'sort' => [
        // Set the default sort by name ASC and created_at DESC.
        'defaultOrder' => [
            'name' => SORT_ASC,
            'created_at' => SORT_DESC
        ]
    ],
    'pagination' => [
        'pageSize' => 20,
    ],
]);

// get the posts in the current page
$posts = $provider->getModels();
```

8.1.2 Array data provider

ArrayDataProvider implements a data provider based on a data array.

The yii\data\ArrayDataProvider::$allModels property contains all data models that may be sorted and/or paginated. ArrayDataProvider will provide the data after sorting and/or pagination. You may configure the yii\data\ArrayDataProvider::$sort and yii\data\ArrayDataProvider ::$pagination properties to customize the sorting and pagination behaviors.

Elements in the yii\data\ArrayDataProvider::$allModels array may be either objects (e.g. model objects) or associative arrays (e.g. query results of DAO). Make sure to set the yii\data\ArrayDataProvider::$key property to the name of the field that uniquely identifies a data record or false if you do not have such a field.

Compared to ActiveDataProvider, ArrayDataProvider could be less efficient because it needs to have yii\data\ArrayDataProvider::$allModels ready.

ArrayDataProvider may be used in the following way:

```
$query = new Query();
$provider = new ArrayDataProvider([
    'allModels' => $query->from('post')->all(),
    'sort' => [
        'attributes' => ['id', 'username', 'email'],
    ],
    'pagination' => [
        'pageSize' => 10,
    ],
]);
// get the posts in the current page
$posts = $provider->getModels();
```

> Note: if you want to use the sorting feature, you must configure the sort property so that the provider knows which columns can be sorted.

8.1.3 SQL data provider

SqlDataProvider implements a data provider based on a plain SQL statement. It provides data in terms of arrays, each representing a row of query result.

Like other data providers, SqlDataProvider also supports sorting and pagination. It does so by modifying the given yii\data\SqlDataProvider:: $sql statement with "ORDER BY" and "LIMIT" clauses. You may configure the yii\data\SqlDataProvider::$sort and yii\data\SqlDataProvider ::$pagination properties to customize sorting and pagination behaviors.

SqlDataProvider may be used in the following way:

```
$count = Yii::$app->db->createCommand('
    SELECT COUNT(*) FROM user WHERE status=:status
', [':status' => 1])->queryScalar();
```

```
$dataProvider = new SqlDataProvider([
    'sql' => 'SELECT * FROM user WHERE status=:status',
    'params' => [':status' => 1],
    'totalCount' => $count,
    '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',
            ],
        ],
    ],
    'pagination' => [
        'pageSize' => 20,
    ],
]);

// get the user records in the current page
$models = $dataProvider->getModels();
```

> Note: if you want to use the pagination feature, you must configure the `yii\data\SqlDataProvider::$totalCount` property to be the total number of rows (without pagination). And if you want to use the sorting feature, you must configure the `yii\data\SqlDataProvider::$sort` property so that the provider knows which columns can be sorted.

8.1.4 Implementing your own custom data provider

TBD

8.2 Data widgets

> Note: This section is under development.

8.2.1 ListView

8.2.2 DetailView

DetailView displays the detail of a single data `yii\widgets\DetailView::$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 of `\yii\base\Model` or an associative array.

DetailView uses the `yii\widgets\DetailView::$attributes` property to determines which model attributes should be displayed and how they should be formatted.

A typical usage of DetailView is as follows:

```
echo DetailView::widget([
    'model' => $model,
    'attributes' => [
        'title',               // title attribute (in plain text)
        'description:html',    // description attribute in HTML
        [                      // the owner name of the model
            'label' => 'Owner',
            'value' => $model->owner->name,
        ],
    ],
]);
```

8.2.3 GridView

Data grid or GridView is one of the most powerful Yii widgets. It is extremely useful if you need to quickly build admin section of the system. It takes data from data provider and renders each row using a set of columns presenting data in a 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 expression of attributes or static text).

Grid view supports both sorting and pagination of the data items. The sorting and pagination can be done in AJAX mode or normal page request. A benefit of using GridView is that when the user disables JavaScript, the sorting and pagination automatically degrade to normal page requests and are still functioning as expected.

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 data provider. The displayed table is equipped with sorting and pagination functionality.

Grid columns

Yii grid consists of a number of columns. Depending on column type and settings these are able to present data differently.

These are defined in the columns part of GridView config like the following:

```
echo GridView::widget([
    'dataProvider' => $dataProvider,
    'columns' => [
        ['class' => 'yii\grid\SerialColumn'],
        // A simple column defined by the data contained in $dataProvider.
        // Data from model's column1 will be used.
        'id',
        'username',
        // More complex one.
        [
            'class' => 'yii\grid\DataColumn', // can be omitted, default
            'value' => function ($data) {
                return $data->name;
            },
        ],
    ],
]);
```

Note that if columns part of config isn't specified, Yii tries to show all possible data provider model columns.

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
        ],
```

Additionally 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 there some common options you can set while configuring grid columns.

- `header` allows to set content for header row.
- `footer` allows to set content for footer row.
- `visible` is the column should be visible.
- `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, $grid) {
    return 'a string';
}
```

You may specify various container HTML options passing arrays to:

- `headerOptions`
- `contentOptions`
- `footerOptions`
- `filterOptions`

Data column Data column is for displaying and sorting data. It is default column type so specifying class could be omitted when using it.

TBD

Action column 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:

- `controller` is the ID of the controller that should handle the actions. If not set, it will use the currently active controller.
- `template` 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`. 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. Default is `{view} {update} {delete}`.
- `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) {
    // return the button HTML code
}
```

In the code above `$url` is the URL that the column creates for the button, and `$model` is the model object being rendered for the current row.

- `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()`.

Checkbox column CheckboxColumn displays a column of checkboxes.

To add a CheckboxColumn to the `yii\grid\GridView`, add it to the `yii\grid\GridView::$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 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

- `https://github.com/yiisoft/yii2/issues/1576`

Filtering data

For filtering data the GridView needs a model that takes the input from the filtering form and adjusts the query of the dataProvider to respect the search criteria. A common practice when using active records is to create a search Model class that extends from the active record class. This class then defines the validation rules for the search and provides a `search()` method that will return the data provider.

To add search capability for the `Post` model we can create `PostSearch` like in 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 seach 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->name])
              ->andFilterWhere(['like', 'creation_date', $this->
    creation_date]);

        return $dataProvider;
    }
}
```

You can use this function in the controller to get the dataProvider for the GridView:

```
$searchModel = new PostSearch();
$dataProvider = $searchModel->search($_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,
]);
```

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's author's name instead of just his `id`. You do this by defining the attribute name in 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']); }]);
// 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: For more information on `joinWith` and the queries performed in the background, check the active record docs on eager and lazy loading.

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. 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 individual sorting and pagination. You do so by setting the `yii\data\Sort::sortParam` and `yii\data\Pagination::pageParam` of the dataProviders `yii\data\BaseDataProvider::$sort` and `yii\data\BaseDataProvider::$pagination` instance.

Assume we want to list `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

TBD

8.3 Theming

> Note: This section is under development.

A theme is a directory of view and layout files. Each file of the theme overrides corresponding file of an application when rendered. A single application may use multiple themes and each may provide totally different experience. At any time only one theme can be active.

> Note: Themes usually do not meant to be redistributed since views are too application specific. If you want to redistribute customized look and feel consider CSS and JavaScript files in form of asset bundles instead.

8.3.1 Configuring a theme

Theme configuration is specified via `view` component of the application. In order to set up a theme to work with basic application views the following should be in your application config file:

```
'components' => [
    'view' => [
        'theme' => [
            'pathMap' => ['@app/views' => '@app/themes/basic'],
            'baseUrl' => '@web/themes/basic',
        ],
    ],
],
```

In the above `pathMap` defines a map of original paths to themed paths while `baseUrl` defines base URL for resources referenced from theme files.

In our case `pathMap` is `['@app/views' => '@app/themes/basic']`. That means that every view in `@app/views` will be first searched under `@app/themes/basic` and if a view exists in the theme directory it will be used instead of the original view.

For example, with a configuration above a themed version of a view file `@app/views/site/index.php` will be `@app/themes/basic/site/index.php`. It basically replaces `@app/views` in `@app/views/site/index.php` with `@app/themes/basic`.

Theming modules

In order to theme modules `pathMap` may look like the following:

```
'components' => [
    'view' => [
        'theme' => [
            '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` with `@app/themes/basic/modules/blog/views/comment/index.php`.

Theming widgets

In order to theme a widget view located at `@app/widgets/currency/views/index.php` you need the following config for view component theme:

```
'components' => [
    'view' => [
        'theme' => [
            'pathMap' => ['@app/widgets' => '@app/themes/basic/widgets'],
        ],
    ],
],
```

With the config above you can create themed version of `@app/widgets/currency/index.php` view in `@app/themes/basic/widgets/currency/index.php`.

8.3.2 Using multiple paths

It is possible to map a single path to multiple theme paths. For example,

```
'pathMap' => [
    '@app/views' => [
        '@app/themes/christmas',
        '@app/themes/basic',
    ],
]
```

In this case, the view will be searched in `@app/themes/christmas/site/index.php` then if it's not found it will check `@app/themes/basic/site/index.php`. If there's no view there as well application view will be used.

This ability is especially useful if you want to temporary or conditionally override some views.

Chapter 9

Security

9.1 Authentication

Note: This section is under development.

Authentication is the act of verifying who a user is, and is the basis of the login process. Typically, authentication uses the combination of an identifier–a username or email address–and a password. The user submits these values through a form, and the application then compares the submitted information against that previously stored (e.g., upon registration).

In Yii, this entire process is performed semi-automatically, leaving the developer to merely implement `yii\web\IdentityInterface`, the most important class in the authentication system. Typically, implementation of `IdentityInterface` is accomplished using the `User` model.

You can find a fully featured example of authentication in the advanced application template. Below, only the interface methods are listed:

```
class User extends ActiveRecord implements IdentityInterface
{
    // ...

    /**
     * Finds an identity by the given ID.
     *
     * @param string|integer $id the ID to be looked for
     * @return IdentityInterface|null the identity object that matches the
    given ID.
     */
    public static function findIdentity($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 $authKey
     * @return boolean if auth key is valid for current user
     */
    public function validateAuthKey($authKey)
    {
        return $this->getAuthKey() === $authKey;
    }
}
```

Two of the outlined methods are simple: `findIdentity` is provided with an ID value and returns a model instance associated with that ID. The `getId` method returns the ID itself. Two of the other methods–`getAuthKey` and `validateAuthKey`–are used to provide extra security to the "remember me" cookie. The `getAuthKey` method should return a string that is unique for each user. You can create reliably create a unique string using `Security::generateRandomKey()`. It's a good idea to also save this as part of the user's record:

```
public function beforeSave($insert)
{
    if (parent::beforeSave($insert)) {
        if ($this->isNewRecord) {
            $this->auth_key = Security::generateRandomKey();
        }
        return true;
    }
    return false;
}
```

The `validateAuthKey` method just needs to compare the `$authKey` variable, passed as parameter (itself retrieved from a cookie), with the value fetched from database.

9.2 Authorization

> Note: This section is under development.

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).

9.2.1 Access Control Filter

Access Control Filter (ACF) is a simple authorization method that is best used by applications that only need some simple access control. As its name indicates, ACF is an action filter that can be attached to a controller or a module as a behavior. ACF will check a set of `yii\filters\AccessControl::rules` to make sure the current user can access the requested action.

The code below shows how to use ACF which is implemented as `yii\filters\AccessControl`:

```
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 `login`, `logout` and `signup` actions. The `rules` option specifies the `yii\filters\AccessRule`, which reads as follows:

- Allow all guest (not yet authenticated) users to access 'login' and 'signup' actions. The `roles` option contains a question mark `?` which is a special token recognized as "guests".
- Allow authenticated users to access 'logout' action. The `@` character is another special token recognized as authenticated users.

When ACF performs authorization check, it will examine the rules one by one from top to bottom until it finds a match. The `allow` value of the matching rule will then be used to judge if the user is authorized. If none of the rules matches, it means the user is NOT authorized and ACF will stop further action execution.

By default, ACF does only of the followings when it determines a user is not authorized to access the current action:

- If the user is a guest, it will call `yii\web\User::loginRequired()`, which may redirect the 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:

```
[
    'class' => AccessControl::className(),
    'denyCallback' => function ($rule, $action) {
        throw new \Exception('You are not allowed to access this page');
    }
]
```

`yii\filters\AccessRule` 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`: specifies whether this is an "allow" or "deny" rule.
- `yii\filters\AccessRule::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`: specifies which controllers this rule matches. This should be an array of controller IDs. 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`: 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 requires RBAC (to be described in the next section), and `yii\web\User::can()` will be called. If this option is empty or not set, it means this rule applies to all roles.
- `yii\filters\AccessRule::ips`: specifies which `yii\web\Request::userIP` 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`: specifies which request method (e.g. `GET`, `POST`) this rule matches. The comparison is case-insensitive.
- `yii\filters\AccessRule::matchCallback`: specifies a PHP callable that should be called to determine if this rule should be applied.
- `yii\filters\AccessRule::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 31
    st
    public function actionSpecialCallback()
    {
        return $this->render('happy-halloween');
    }
}
```

9.2.2 Role based access control (RBAC)

Role-Based Access Control (RBAC) provides a simple yet powerful centralized access control. Please refer to the Wiki article[1] for details about comparing RBAC with other more traditional access control schemes.

Yii implements a General Hierarchical RBAC, following the NIST RBAC model[2]. It provides the RBAC functionality through the `yii\rbac\ManagerInterface` 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

[1] http://en.wikipedia.org/wiki/Role-based_access_control
[2] http://csrc.nist.gov/rbac/sandhu-ferraiolo-kuhn-00.pdf

the more special *tree* hierarchy. While a role can contain a permission, it is not true vice versa.

Configuring RBAC Manager

Before we set off to define authorization data and perform access checking, we need to configure the `yii\base\Application::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 database. You may consider using the former if your application does not require very dynamic role and permission management.

The following code shows how to configure `authManager` in the application configuration:

```
return [
    // ...
    'components' => [
        'authManager' => [
            'class' => 'yii\rbac\PhpManager',
        ],
        // ...
    ],
];
```

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.

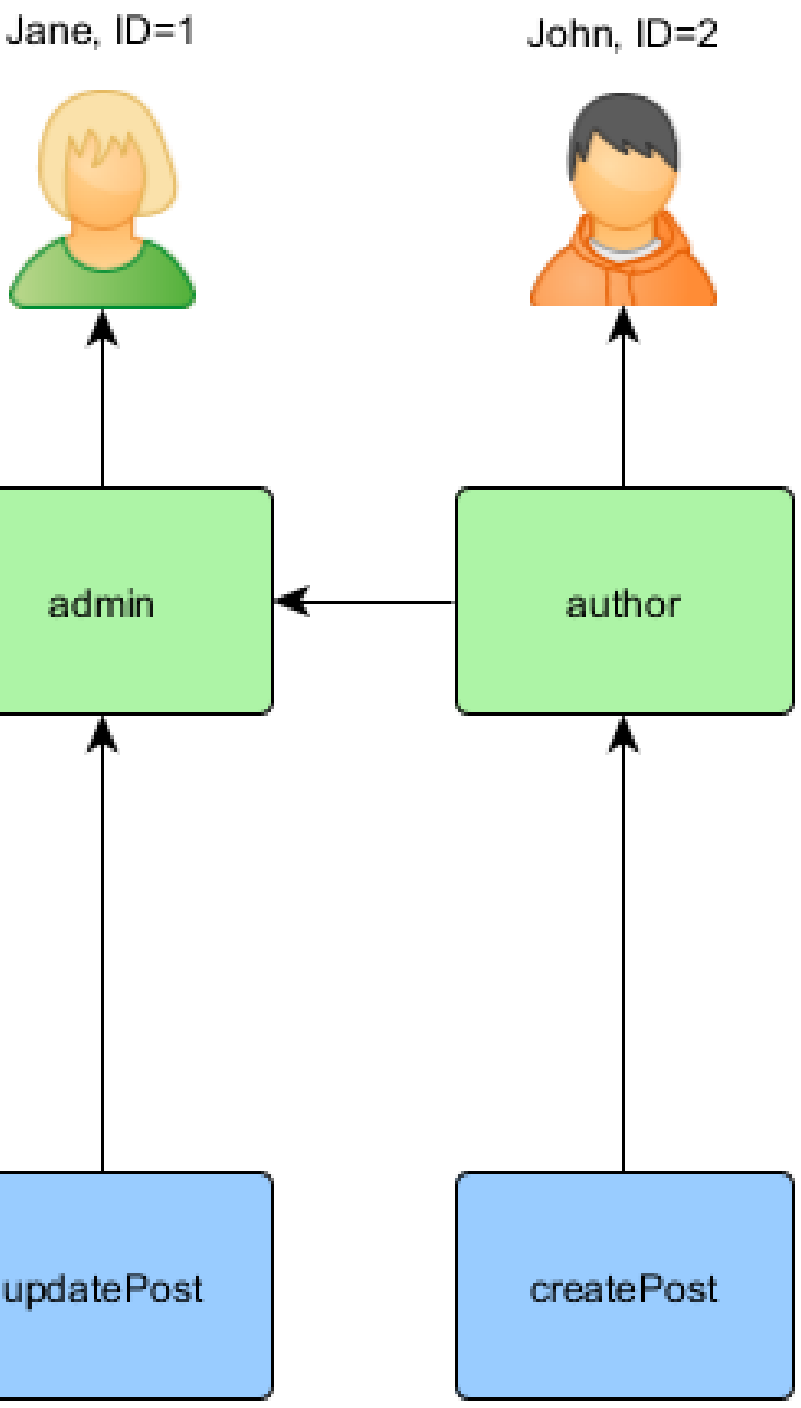
Depending on authorization flexibility requirements the tasks above could be done in different ways.

If your persmissions 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
namespace app\commands;
```

```
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);
    }
}
```

After executing the command 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 you in advanced application 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`.

> Tip: By default, `yii\rbac\PhpManager` stores RBAC data in the file `@app/data/rbac.php`. Sometimes when you want to make some minor changes to the RBAC data, you may directly edit this file.

Using Rules

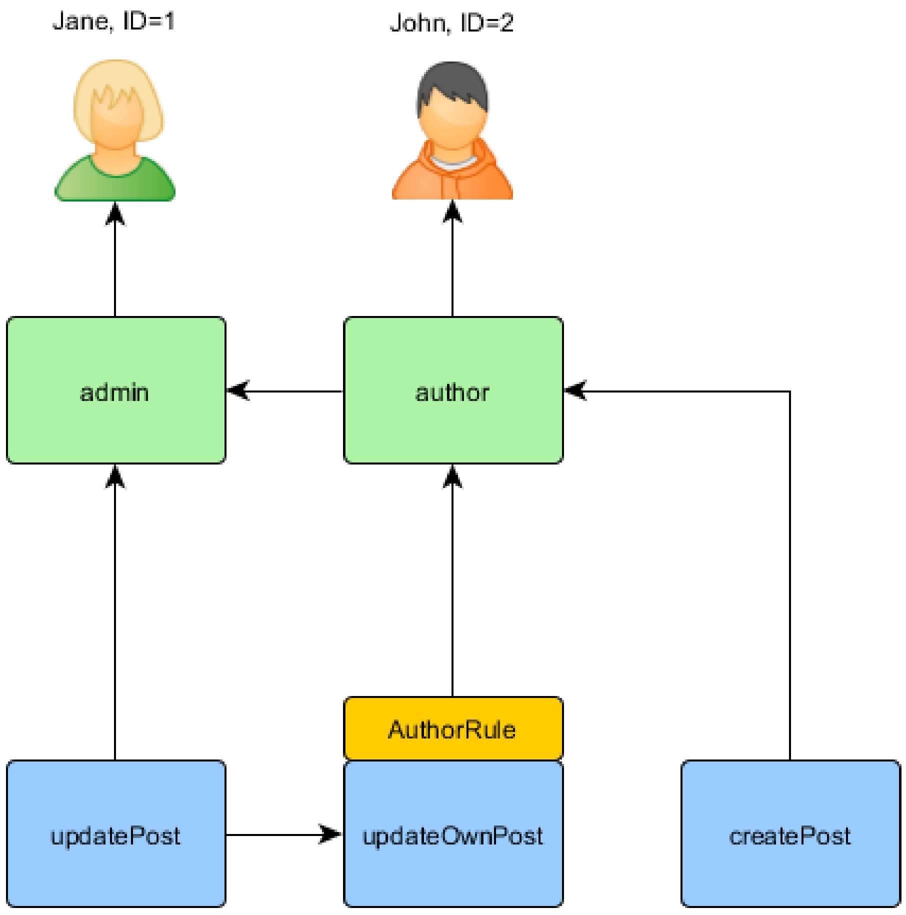
As aforementioned, rules add additional constraint to roles and permissions. A rule is a class extending from `yii\rbac\Rule`. It must implement the `yii\rbac\Rule::execute()` method. In the hierarchy we've created previously author cannot edit his own post. Let's fix it. First we need a rule to verify 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|integer $user the user ID.
     * @param Item $item the role or permission that this rule is associated
     with
     * @param array $params parameters passed to ManagerInterface::
    checkAccess().
     * @return boolean a value indicating whether the rule permits the role
    or permission it is associated with.
     */
    public function execute($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:

```
// add the rule
$rule = new \app\rbac\AuthorRule;
$auth->add($rule);

// add the "updateOwnPost" permission and associate the rule with it.
$updateOwnPost = $this->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've got the following hierarchy:

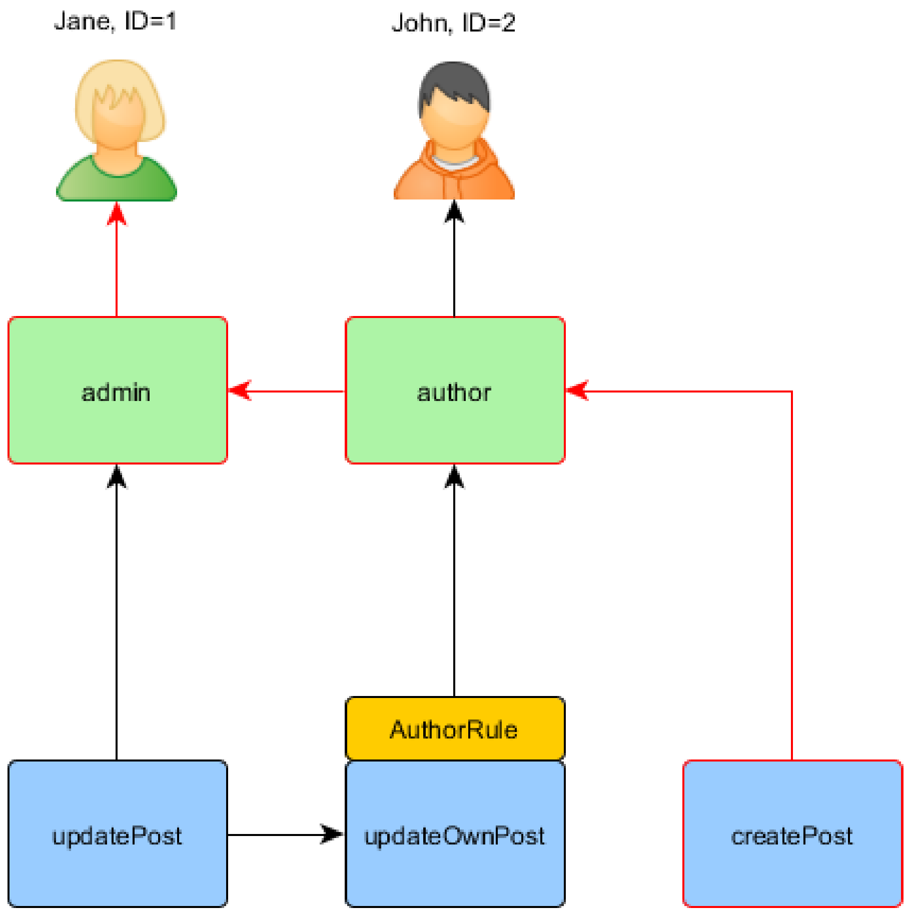


Access Check

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
}
```

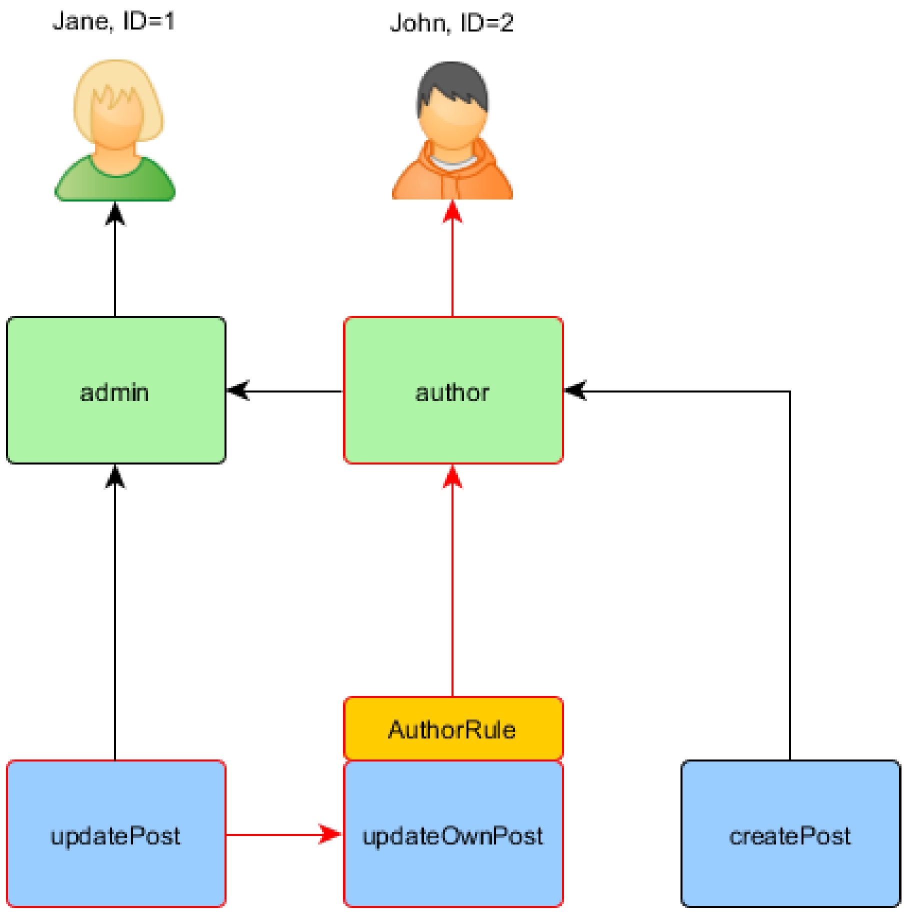
If the current user is Jane with ID=1 we're starting at `createPost` and trying to get to `Jane`:



In order to check if user can update post we need to pass an extra parameter that is required by the `AuthorRule` described before:

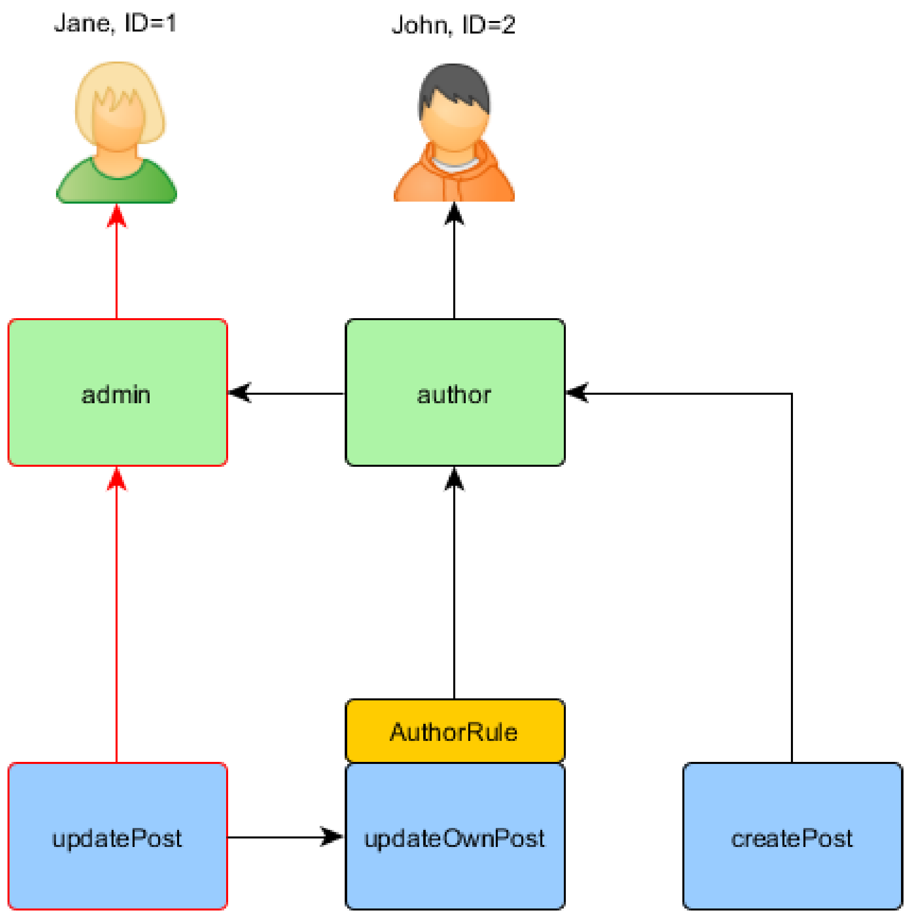
```
if (\Yii::$app->user->can('updatePost', ['post' => $post])) {
    // update post
}
```

Here's what happens if current user is John:



We're starting with the `updatePost` and going through `updateOwnPost`. In order to pass it `AuthorRule` should return `true` from its `execute` method. The method receives its `$params` from `can` method call so the value is `['post' => $post]`. If everything is OK we're getting to `author` that is assigned to John.

In case of Jane it is a bit simpler since she's an admin:



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 a RBAC role, you can use the default role feature to automatically assign each user to a 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 create set up the RBAC data as follows,

```
namespace app\rbac;

use Yii;
use yii\rbac\Rule;

/**
 * Checks if authorID matches user passed via params
 */
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;
    }
}

$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.

9.3 Security

> Note: This section is under development.

Good security is vital to the health and success of any application. Unfortunately, many developers cut corners when it comes to security, either due to a lack of understanding or because implementation is too much of a hurdle. To make your Yii powered application as secure as possible, Yii has included several excellent and easy to use security features.

9.3.1 Hashing and verifying 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 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[3]. 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:

```
$hash = \yii\helpers\Security::generatePasswordHash($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:

[3]http://php.net/manual/en/function.crypt.php

```
use yii\helpers\Security;
if (Security::validatePassword($password, $hash)) {
    // all good, logging user in
} else {
    // wrong password
}
```

9.3.2 Generating Pseudorandom data

Pseudorandom data is useful in many situations. For example when resetting a password via email you need to generate a token, save it to the database, and send it via email to end user which in turn will allow them to prove ownership of that account. It is very important that this token be unique and hard to guess, else there is a possibility and attacker can predict the token's value and reset the user's password.

Yii security helper makes generating pseudorandom data simple:

```
$key = \yii\helpers\Security::generateRandomKey();
```

Note that you need to have the `openssl` extension installed in order to generate cryptographically secure random data.

9.3.3 Encryption and decryption

Yii provides convenient helper functions that allow you to encrypt/decrypt data using a secret key. The data is passed through and encryption function so that only the person which has the secret key will be able to decrypt it. For example, we need to store some information in our database but we need to make sure only the user which has the secret key can view it (even if the application database is compromised):

```
// $data and $secretKey are obtained from the form
$encryptedData = \yii\helpers\Security::encrypt($data, $secretKey);
// store $encryptedData to database
```

Subsequently when user wants to read the data:

```
// $secretKey is obtained from user input, $encryptedData is from the
    database
$data = \yii\helpers\Security::decrypt($encryptedData, $secretKey);
```

9.3.4 Confirming data integrity

There are situations in which you need to verify that your data hasn't been tampered with by a third party or even corrupted in some way. Yii provides an easy way to confirm data integrity in the form of two helper functions.

Prefix the data with a hash generated from the secret key and data

```
// $secretKey our application or user secret, $genuineData obtained from a
    reliable source
$data = \yii\helpers\Security::hashData($genuineData, $secretKey);
```

Checks if the data integrity has been compromised

```
// $secretKey our application or user secret, $data obtained from an
    unreliable source
$data = \yii\helpers\Security::validateData($data, $secretKey);
```

todo: XSS prevention, CSRF prevention, cookie protection, refer to 1.1 guide

You also can disable CSRF validation per controller and/or action, 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);
    }
}
```

9.3.5 Securing Cookies

- validation
- httpOnly is default

9.3.6 See also

- Views security

Error: not existing file: security-auth-clients.md

Error: not existing file: security-best-practices.md

Chapter 10

Caching

10.1 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

10.2 Data Caching

Data caching is about storing some PHP variable in cache and retrieving it later from cache. It is also the foundation for more advanced caching features, such as query caching and content 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

    // store $data in cache so that it can be retrieved next time
    $cache->set($key, $data);
}

// $data is available here
```

10.2.1 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[1] 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`:

```
'components' => [
    'cache' => [
```

[1] http://memcached.org/

```
        '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[2] 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 particular suitable to cache large chunk of data, such as page content.

- `yii\caching\MemCache`: uses PHP memcache[3] and memcached[4] 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[5] key-value store (redis version 2.6.12 or higher is required).

[2] http://php.net/manual/en/book.apc.php
[3] http://php.net/manual/en/book.memcache.php
[4] http://php.net/manual/en/book.memcached.php
[5] http://redis.io/

- `yii\caching\WinCache`: uses PHP WinCache[6] (see also[7]) extension.
- `yii\caching\XCache`: uses PHP XCache[8] extension.
- Zend Data Cache[9] 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).

10.2.2 Cache APIs

All cache components have the same base class `yii\caching\Cache` and thus support the following APIs:

- `yii\caching\Cache::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()`: stores a data item identified by a key in cache.
- `yii\caching\Cache::add()`: stores a data item identified by a key in cache if the key is not found in the cache.
- `yii\caching\Cache::mget()`: retrieves multiple data items from cache with the specified keys.
- `yii\caching\Cache::mset()`: stores multiple data items in cache. Each item is identified by a key.
- `yii\caching\Cache::madd()`: 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()`: returns a value indicating whether the specified key is found in the cache.
- `yii\caching\Cache::delete()`: removes a data item identified by a key from the cache.

[6]http://iis.net/downloads/microsoft/wincache-extension
[7]http://php.net/manual/en/book.wincache.php
[8]http://xcache.lighttpd.net/
[9]http://files.zend.com/help/Zend-Server-6/zend-server.htm#data_cache_component.htm

- `yii\caching\Cache::flush()`: removes all data items from the cache.

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::mget()` and `yii\caching\Cache::madd()` 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 liked an array. The followings 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.

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()` 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()` 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
}
```

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()` call should return false.

Cache dependencies are represented as objects of `yii\caching\Dependency` descendant classes. When you call `yii\caching\Cache::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 is 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\GroupDependency`: marks a cached data item with a group name. You may invalidate the cached data items with the same group name all at once by calling `yii\caching\GroupDependency::invalidate()`.

10.2.3 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` and a valid `cache` application component. The basic usage of query caching is as follows, assuming `$db` is a `yii\db\Connection` instance:

```
$duration = 60;     // cache query results for 60 seconds.
$dependency = ...;  // optional dependency

$db->beginCache($duration, $dependency);

// ...performs DB queries here...

$db->endCache();
```

As you can see, any SQL queries in between the `beginCache()` and `endCache()` calls will be cached. If the result of the same query is found valid in the cache, the query will be skipped and the result will be served from the cache instead.

Query caching can be used for DAO as well as ActiveRecord.

> Info: Some DBMS (e.g. MySQL[10]) 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.

Configurations

Query caching has two two configurable options through `yii\db\Connection`:

[10]http://dev.mysql.com/doc/refman/5.1/en/query-cache.html

- `yii\db\Connection::queryCacheDuration`: this represents the number of seconds that a query result can remain valid in the cache. The duration will be overwritten if you call `yii\db\Connection::beginCache()` with an explicit duration parameter.
- `yii\db\Connection::queryCache`: this represents the ID of the cache application component. It defaults to `'cache'`. Query caching is enabled only when there is a valid cache application component.

Limitations

Query caching does not work with query results that contain resource handles. For example, when using the `BLOB` column type in some DBMS, the query result will return a resource handle 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.

10.3 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()` and `yii\base\View::endCache()` calls. If the content is found in the cache, `yii\base\View::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()` 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.

10.3.1 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()` 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. 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 0, which means the cached content will never expire.

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 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` 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` option, like the following:

```
if ($this->beginCache($id, ['enabled' => Yii::$app->request->isGet])) {

    // ... generate content here ...

    $this->endCache();
}
```

10.3.2 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.

10.3.3 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,

```php
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()` 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.

10.4 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.

10.5 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` only works for `GET` and `HEAD` requests. It can handle three kinds of cache-related HTTP headers for these requests:

- `yii\filters\HttpCache::lastModified`
- `yii\filters\HttpCache::etagSeed`
- `yii\filters\HttpCache::cacheControlHeader`

10.5.1 `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 $params the value of the "params" property
 * @return integer a UNIX timestamp representing the page modification time
 */
function ($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.

10.5.2 `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 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 compliant to RFC 2616, section 13.3.4[11], `HttpCache` will send out both `ETag` and `Last-Modified` headers if they are both configured. Consequently, both will be used for cache validation if sent by the client.

[11] http://tools.ietf.org/html/rfc2616#section-13.3.4

10.5.3 `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
```

10.5.4 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()[12] for explanations about these values.

10.5.5 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.

[12]`http://www.php.net/manual/en/function.session-cache-limiter.php`

Chapter 11

RESTful Web Services

11.1 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 (supporting JSON and XML by default) negotiation;
- Customizable object serialization with support for selectable output fields;
- Proper formatting of collection data and validation errors;
- Support for HATEOAS[1];
- 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 `yii\db\ActiveRecord` class `app\models\User` to access the user data.

[1] `http://en.wikipedia.org/wiki/HATEOAS`

11.1.1 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`. By specifying `yii\rest\ActiveController::modelClass` as `app\models\User`, the controller knows what model can be used for fetching and manipulating data.

11.1.2 Configuring URL Rules

Then, modify the configuration about 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.

11.1.3 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`.

  Info: Yii will automatically pluralize controller names for use in endpoints.

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
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,
        ...
    },
    ...
]
```

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
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/xml

<?xml version="1.0" encoding="UTF-8"?>
<response>
    <item>
        <id>1</id>
        ...
    </item>
    <item>
        <id>2</id>
        ...
    </item>
    ...
</response>
```

> 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 are 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 Response Formatting section.

11.1.4 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` 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.

11.2 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.

11.2.1 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`. The first step is what you should mainly focus when developing a resource class.

By overriding `yii\base\Model::fields()` and/or `yii\base\Model::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 extraFields()
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 backward 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 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 nothing, 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,
        }
    },
    ...
]
```

11.2.2 Links

HATEOAS[2], an abbreviation for Hypermedia as the Engine of Application State, promotes that RESTful APIs should return information that allow 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()` which should return a list of `yii\web\Link`. Typically, you should return at least the `self` link representing the URL to the resource object itself. For example,

```
use yii\db\ActiveRecord;
use yii\web\Link;
use yii\web\Linkable;
use yii\helpers\Url;

class User extends ActiveRecord implements Linkable
{
    public function getLinks()
    {
        return [
            Link::REL_SELF => Url::to(['user', 'id' => $this->id], true),
        ];
    }
}
```

When a `User` 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": "https://example.com/users/100"
    ]
}
```

11.2.3 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:

[2]http://en.wikipedia.org/wiki/HATEOAS

```
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.

11.3 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 regarding to the requested action and resource.

11.3.1 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` and the `yii\web\Response` will handle the conversion from the original data to the requested format. For example,

```
public function actionView($id)
{
    return User::findOne($id);
}
```

11.3.2 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`: supports content negotiation, to be explained in the Response Formatting section;
- `yii\filters\VerbFilter`: supports HTTP method validation;
- `yii\filters\AuthMethod`: supports user authentication, to be explained in the Authentication section;

- `yii\filters\RateLimiter`: supports rate limiting, to be explained in the Rate Limiting section.

These named filters are declared in the `yii\rest\Controller::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;
}
```

11.3.3 Extending `ActiveController`

If your controller class extends from `yii\rest\ActiveController`, you should set its `yii\rest\ActiveController::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`: list resources page by page;
- `yii\rest\ViewAction`: return the details of a specified resource;
- `yii\rest\CreateAction`: create a new resource;
- `yii\rest\UpdateAction`: update an existing resource;
- `yii\rest\DeleteAction`: delete the specified resource;
- `yii\rest\OptionsAction`: return the supported HTTP methods.

All these actions are declared through the `yii\rest\ActiveController::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()` method like the following,

```
/**
 * Checks the privilege of the current user.
 *
 * This method should be overridden to check whether the current user has
    the privilege
 * 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 specific 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
}
```

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.

11.4 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 endpoints. This is because `yii\rest\UrlRule` automatically pluralizes controller IDs for them to use in endpoints. You may disable this behavior by setting `yii\rest\UrlRule::pluralize` to be false, or if you want to use some special names you may configure the `yii\rest\UrlRule ::controller` property.

11.5 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[3].
2. Convert resource objects into arrays, as described in the Resources section. This is done by `yii\rest\Serializer`.

[3]http://en.wikipedia.org/wiki/Content_negotiation

3. Convert arrays into a string in the format as determined by the content negotiation step. This is done by `yii\web\ResponseFormatterInterface` registered with the `yii\web\Response::formatters` application component.

11.5.1 Content Negotiation

Yii supports content negotiation via the `yii\filters\ContentNegotiator` filter. The the RESTful API base controller class `yii\rest\Controller` is equipped with this filter under the name of `contentNegotiator`. The filer 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` 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 property of the contentNegotiator filter like the following in your API controller classes:

```
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.

11.5.2 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\ArrayableInterface 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": "http://localhost/users?page=1",
        "next": "http://localhost/users?page=2",
        "last": "http://localhost/users?page=50"
    },
    "_meta": {
        "totalCount": 1000,
        "pageCount": 50,
        "currentPage": 1,
        "perPage": 20
    }
}
```

11.6 Authentication

Unlike Web applications, RESTful APIs should be 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, **the API requests should always be sent via HTTPS to prevent from man-in-the-middle (MitM) attacks**.

There are different ways to send an access token:

- HTTP Basic Auth[4]: the access token is sent as the username. This

[4]http://en.wikipedia.org/wiki/Basic_access_authentication

is 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[5]: the access token is obtained by the consumer from an authorization server and sent to the API server via HTTP Bearer Tokens[6], 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 two steps:

1. Specify which authentication methods you plan to use by configuring the `authenticator` behavior in your REST controller classes.
2. Implement `yii\web\IdentityInterface::findIdentityByAccessToken()` in your `yii\web\User::identityClass`.

For example, to use HTTP Basic Auth, you may configure `authenticator` 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()
{
```

[5]http://oauth.net/2/
[6]http://tools.ietf.org/html/rfc6750

```
    $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).

11.6.1 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\Controller::checkAccess()` method to perform authorization check. The method will be called by the built-in actions provided by `yii\rest\ActiveController`.

11.7 Rate Limiting

To prevent abuse, you should consider adding rate limiting to your APIs. For example, you may 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 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` should implement `yii\filters\RateLimitInterface`. This interface requires implementation of the following three methods:

- `getRateLimit()`: returns the maximum number of allowed requests and the time period, e.g., `[100, 600]` means 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 is checked last time.
- `saveAllowance()`: saves the number of remaining requests allowed and the current UNIX timestamp.

You may use two columns in the user table to record the allowance and timestamp information. And `loadAllowance()` and `saveAllowance()` can then be implementation by reading and saving the values of the two columns corresponding to the current authenticated user. To improve performance, you may also consider storing these information in cache or some NoSQL storage.

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 thrown a `yii\web\TooManyRequestsHttpException` if 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, like shown in the above code example.

11.8 Versioning

Your APIs should be versioned. Unlike Web applications which you have full control on both client side and server side code, for APIs you usually do not have control of the client code that consumes the APIs. Therefore, backward compatibility (BC) of the APIs should be maintained whenever possible, and if some BC-breaking changes must be introduced to the APIs, you should bump up the version number. You may refer to Semantic Versioning[7] for more information about designing the version numbers of your APIs.

Regarding how to implement API versioning, a common practice is to embed the version number in the API URLs. For example, `http://example.com/v1/users` stands for `/users` API of version 1. Another method of API versioning which gains momentum recently is to put version numbers in the HTTP request headers, typically through the `Accept` header, like the following:

```
// via a parameter
Accept: application/json; version=v1
// via a vendor content type
Accept: application/vnd.company.myapp-v1+json
```

Both methods have pros and cons, and there are a lot of debates about them. Below we describe a practical strategy of API versioning that is kind of 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, it should include the resource classes and the controller classes serving for 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:

[7]`http://semver.org/`

```
api/
    common/
        controllers/
            UserController.php
            PostController.php
        models/
            User.php
            Post.php
    modules/
        v1/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
        v2/
            controllers/
                UserController.php
                PostController.php
            models/
                User.php
                Post.php
```

Your application configuration would look like:

```
return [
    'modules' => [
        'v1' => [
            'basePath' => '@app/modules/v1',
        ],
        'v2' => [
            'basePath' => '@app/modules/v2',
        ],
    ],
    '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, `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.

Using modules, code for different major versions can be well isolated. And it is still possible to reuse code across modules via common base classes and other shared classes.

To deal with minor version numbers, you may take advantage of the content negotiation feature provided by the `yii\filters\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.

Since minor versions require maintaining backward compatibility, hopefully there are not much version checks in your code. Otherwise, chances are that you may need to create a new major version.

11.9 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 wrong has happened. If you can identify the cause of the error (e.g. the requested resource does not exist), you should consider throwing an exception with a proper HTTP status code (e.g. `yii\web\NotFoundHttpException` representing a 404 HTTP status code). Yii will send the response with the corresponding HTTP status code and text. It will also include in the response body the serialized representation of the exception. 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

{
    "type": "yii\\web\\NotFoundHttpException",
    "name": "Not Found Exception",
    "message": "The requested resource was not found.",
    "code": 0,
    "status": 404
}
```

The following list summarizes the HTTP status code 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 is handled successfully and the response contains no body content (like a `DELETE` request).
- `304`: Resource was not modified. You can use the cached version.
- `400`: Bad request. This could be caused by various reasons from the user side, such as invalid JSON data in the request body, 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 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 is rejected due to rate limiting.
- `500`: Internal server error. This could be caused by internal program errors.

Chapter 12

Development Tools

12.1 Debug toolbar and debugger

> Note: This section is under development.

Yii2 includes a handy toolbar, and built-in debugger, for faster development and debugging of your applications. The toolbar displays information about the currently opened page, while the debugger can be used to analyze data you've previously collected (i.e., to confirm the values of variables).

Out of the box these tools allow you to:

- Quickly get the framework version, PHP version, response status, current controller and action, performance info and more via toolbar
- Browse the application and PHP configuration
- View the request data, request and response headers, session data, and environment variables
- See, search, and filter the logs
- View any profiling results
- View the database queries executed by the page
- View the emails sent by the application

All of this information will be available per request, allowing you to revisit the information for past requests as well.

12.1.1 Installing and configuring

To enable these features, add these lines to your configuration file to enable the debug module:

```
'bootstrap' => ['debug'],
'modules' => [
    'debug' => 'yii\debug\Module',
]
```

By default, the debug module only works when browsing the website from localhost. If you want to use it on a remote (staging) server, add the parameter `allowedIPs` to the configuration to whitelist your IP:

```
'bootstrap' => ['debug'],
'modules' => [
    'debug' => [
        'class' => 'yii\debug\Module',
        'allowedIPs' => ['1.2.3.4', '127.0.0.1', '::1']
    ]
]
```

If you are using `enableStrictParsing` URL manager option, add the following to your `rules`:

```
'urlManager' => [
    'enableStrictParsing' => true,
    'rules' => [
        // ...
        'debug/<controller>/<action>' => 'debug/<controller>/<action>',
    ],
],
```

> Note: the debugger stores information about each request in the `@runtime/debug` directory. If you have problems using The debugger such as weird error messages when using it or the toolbar not showing up or not showing any requests, check whether the web server has enough permissions to access this directory and the files located inside.

Extra configuration for logging and profiling

Logging and profiling are simple but powerful tools that may help you to understand the execution flow of both the framework and the application. These tools are useful for development and production environments alike.

While in a production environment, you should log only significantly important messages manually, as described in logging guide section. It hurts performance too much to continue to log all messages in production.

In a development environment, the more logging the better, and it's especially useful to record the execution trace.

In order to see the trace messages that will help you to understand what happens under the hood of the framework, you need to set the trace level in the configuration file:

```
return [
    // ...
    'components' => [
        'log' => [
            'traceLevel' => YII_DEBUG ? 3 : 0, // <-- here
```

By default, the trace level is automatically set to `3` if Yii is running in debug mode, as determined by the presence of the following line in your `index.php` file:

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

> Note: Make sure to disable debug mode in production environments since it may have a significant and adverse performance effect. Further, the debug mode may expose sensitive information to end users.

12.1.2 Creating your own panels

Both the toolbar and debugger are highly configurable and customizable. To do so, you can create your own panels that collect and display the specific data you want. Below we'll describe the process of creating a simple custom panel that:

- Collects the views rendered during a request
- Shows the number of views rendered in the toolbar
- Allows you to check the view names in the debugger

The assumption is that you're using the basic application template.

First we need to implement the `Panel` class in `panels/ViewsPanel.php`:

```
<?php
namespace app\panels;

use yii\base\Event;
use yii\base\View;
use yii\base\ViewEvent;
use yii\debug\Panel;


class ViewsPanel extends Panel
{
    private $_viewFiles = [];

    public function init()
    {
        parent::init();
        Event::on(View::className(), View::EVENT_BEFORE_RENDER, function (
    ViewEvent $event) {
            $this->_viewFiles[] = $event->sender->getViewFile();
        });
    }


    /**
     * @inheritdoc
     */
    public function getName()
    {
        return 'Views';
    }

    /**
     * @inheritdoc
     */
    public function getSummary()
    {
        $url = $this->getUrl();
        $count = count($this->data);
        return "<div class=\"yii-debug-toolbar-block\"><a href=\"$url\">
    Views <span class=\"label\">$count</span></a></div>";
    }

    /**
     * @inheritdoc
     */
    public function getDetail()
    {
        return '<ol><li>' . implode('<li>', $this->data) . '</ol>';
    }

    /**
     * @inheritdoc
     */
    public function save()
    {
        return $this->_viewFiles;
    }
}
```

The workflow for the code above is:

1. `init` is executed before any controller action is run. This method is the best place to attach handlers that will collect data during the controller action's execution.

2. `save` is called after controller action is executed. The data returned by this method will be stored in a data file. If nothing is returned by this method, the panel won't be rendered.

3. The data from the data file is loaded into `$this->data`. For the toolbar, this will always represent the latest data, For the debugger, this

property may be set to be read from any previous data file as well.

4. The toolbar takes its contents from `getSummary`. There, we're showing the number of view files rendered. The debugger uses `getDetail` for the same purpose.

Now it's time to tell the debugger to use the new panel. In `config/web.php`, the debug configuration is modified to:

```
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = [
        'class' => 'yii\debug\Module',
        'panels' => [
            'views' => ['class' => 'app\panels\ViewsPanel'],
        ],
    ];

// ...
```

That's it. Now we have another useful panel without writing much code.

12.2 The Gii code generation tool

> Note: This section is under development.

Yii includes a handy tool, named Gii, that provides rapid prototyping by generating commonly used code snippets as well as complete CRUD controllers.

12.2.1 Installing and configuring

Gii is an official Yii extension. The preferred way to install this extension is through composer[1].

You can either run this command:

```
php composer.phar require --prefer-dist yiisoft/yii2-gii "*"
```

Or you can add this code to the require section of your `composer.json` file:

```
"yiisoft/yii2-gii": "*"
```

Once the Gii extension has been installed, you enable it by adding these lines to your application configuration file:

```
return [
    'bootstrap' => ['gii'],
    'modules' => [
        'gii' => 'yii\gii\Module',
```

[1] http://getcomposer.org/download/

```
        // ...
    ],
    // ...
];
```

You can then access Gii through the following URL:

```
http://localhost/path/to/index.php?r=gii
```

If you have enabled pretty URLs, you may use the following URL:

```
http://localhost/path/to/index.php/gii
```

> Note: if you are accessing gii from an IP address other than localhost, access will be denied by default. To circumvent that default, add the allowed IP addresses to the configuration:
>
> ```
> 'gii' => [
>     'class' => 'yii\gii\Module',
>     'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '
>     192.168.178.20'] // adjust this to your needs
> ],
> ```

Basic application

In basic application template configuration structure is a bit different so Gii should be configured in `config/web.php`:

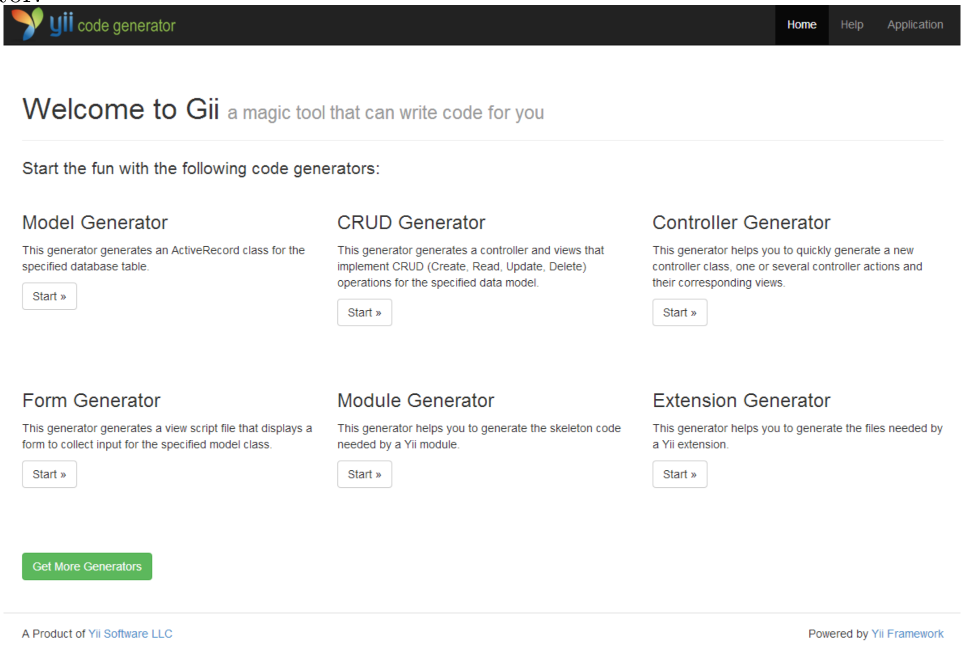
```
// ...
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = 'yii\debug\Module';

    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = 'yii\gii\Module'; // <--- here
}
```

So in order to adjust IP address you need to do it like the following:

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

    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
        'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '192.168.178.20'
    ],
    ];
}
```

12.2.2 How to use it

When you open Gii you first see the entry page that lets you choose a generator.



Welcome to Gii a magic tool that can write code for you

Start the fun with the following code generators:

Model Generator

This generator generates an ActiveRecord class for the specified database table.

CRUD Generator

This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.

Controller Generator

This generator helps you to quickly generate a new controller class, one or several controller actions and their corresponding views.

Form Generator

This generator generates a view script file that displays a form to collect input for the specified model class.

Module Generator

This generator helps you to generate the skeleton code needed by a Yii module.

Extension Generator

This generator helps you to generate the files needed by a Yii extension.

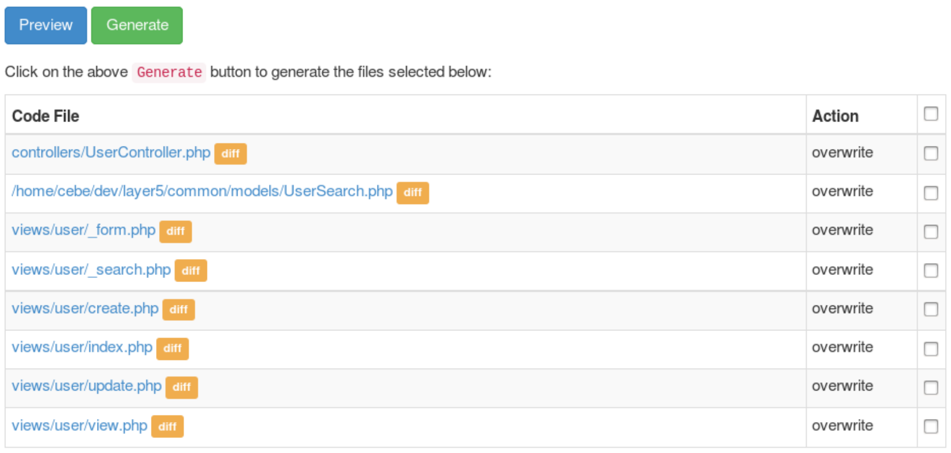
A Product of Yii Software LLC

Powered by Yii Framework

By default there are the following generators available:

- **Model Generator** - This generator generates an ActiveRecord class for the specified database table.
- **CRUD Generator** - This generator generates a controller and views that implement CRUD (Create, Read, Update, Delete) operations for the specified data model.
- **Controller Generator** - This generator helps you to quickly generate a new controller class, one or several controller actions and their corresponding views.
- **Form Generator** - This generator generates a view script file that displays a form to collect input for the specified model class.
- **Module Generator** - This generator helps you to generate the skeleton code needed by a Yii module.
- **Extension Generator** - This generator helps you to generate the files needed by a Yii extension.

After choosing a generator by clicking on the “Start” button you will see a form that allows you to configure the parameters of the generator. Fill out the form according to your needs and press the "Preview" button to get a preview of the code that gii is about to generated. Depending on the generator you chose and whether the files already existed or not, you will get an output similar to what you see in the following picture:



Click on the above `Generate` button to generate the files selected below:

Code File	Action	☐
controllers/UserController.php diff	overwrite	☐
/home/cebe/dev/layer5/common/models/UserSearch.php diff	overwrite	☐
views/user/_form.php diff	overwrite	☐
views/user/_search.php diff	overwrite	☐
views/user/create.php diff	overwrite	☐
views/user/index.php diff	overwrite	☐
views/user/update.php diff	overwrite	☐
views/user/view.php diff	overwrite	☐

Clicking on the file name you can view a preview of the code that will be generated for that file. When the file already exists, gii also provides a diff view that shows what is different between the code that exists and the one that will be generated. In this case you can also choose which files should be overridden and which not.

> Tip: When using the Model Generator to update models after database change, you can copy the code from gii preview and merge the changes with your own code. You can use IDE features like PHPStorms compare with clipboard[2] for this, which allows you to merge in relevant changes and leave out others that may revert your own code.

After you have reviewed the code and selected the files to be generated you can click the "Generate" button to create the files. If all went fine you are done. When you see errors that gii is not able to generate the files you have to adjust directory permissions so that your webserver is able to write to the directories and create the files.

> Note: The code generated by gii is only a template that has to be adjusted to your needs. It is there to help you create new things quickly but it is not something that creates ready to use code. We often see people using the models generated by gii without change and just extend them to adjust some parts of it. This is not how it is meant to be used. Code generated by gii may be incomplete or incorrect and has to be changed to fit your needs before you can use it.

[2]`http://www.jetbrains.com/phpstorm/webhelp/comparing-files.html`

12.2.3 Creating your own templates

Every generator has a form field `Code Template` that lets you choose a template to use for code generation. By default gii only provides one template `default` but you can create your own templates that are adjusted to your needs.

If you open a folder `@app\vendor\yiisoft\yii2-gii\generators`, you'll see six folders of generators. ' + controller - crud

```
+ default
```

- extension
- form
- model
- module

  ```
  This is name generator. If you open any of these folders, you can see
      the folder 'default'. This folder is name of the template.
  ```

Copy folder `@app\vendor\yiisoft\yii2-gii\generators\crud\default` to another location, for example `@app\myTemplates\crud\`. Now open this folder and modify any template to fit your desires, for example, add `errorSummary` in `views\_form.php`:

```
<?php
//...
<div class="<?= Inflector::camel2id(StringHelper::basename($generator->
    modelClass)) ?>-form">

    <?= "<?php " ?>$form = ActiveForm::begin(); ?>
    <?= "<?=" ?> $form->errorSummary($model) ?> <!-- ADDED HERE -->
    <?php foreach ($safeAttributes as $attribute) {
        echo "    <?= " . $generator->generateActiveField($attribute) . "
    ?>\n\n";
    } ?>
//...
```

Now you need to tell GII about our template.The setting is made in the config file:

```
// config/web.php for basic app
// ...
if (YII_ENV_DEV) {
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
        'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '192.168.178.20'
    ],
        'generators' => [ //here
            'crud' => [ //name generator
                'class' => 'yii\gii\generators\crud\Generator', //class
    generator
```

```
                'templates' => [ //setting for out templates
                    'myCrud' => '@app\myTemplates\crud\default', //name
    template => path to template
                ]
            ]
        ],
    ];
}
```

Open the CRUD generator and you will see that in the field `Code Template` of form appeared own template .

12.2.4 Creating your own generators

Open the folder of any generator and you will see two files `form.php` and `Generator.php`. One is the form, the second is the class generator. For create your own generator, you need to create or override these classes in any folder. Again as in the previous paragraph customize configuration:

```
//config/web.php for basic app
//..
if (YII_ENV_DEV) {
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
        'allowedIPs' => ['127.0.0.1', '::1', '192.168.0.*', '192.168.178.20'
    ],
         'generators' => [
            'myCrud' => [
                'class' => 'app\myTemplates\crud\Generator',
                'templates' => [
                    'my' => '@app/myTemplates/crud/default',
                ]
            ]
        ],
    ];
}
```

```
// @app/myTemplates/crud/Generator.php
<?php
namespace app\myTemplates\crud;

class Generator extends \yii\gii\Generator
{
    public function getName()
    {
        return 'MY CRUD Generator';
    }

    public function getDescription()
    {
        return 'My crud generator. The same as a native, but he is mine...';
    }
    // ...
}
```

Open Gii Module and you will see a new generator appears in it.

Error: not existing file: tool-api-doc.md

Chapter 13

Testing

13.1 Testing

TBD

Error: not existing file: test-unit.md

Error: not existing file: test-functional.md

Error: not existing file: test-acceptance.md

13.2 Fixtures

> Note: This section is under development.

Fixtures are 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 objects*. 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.

13.2.1 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
<?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`.

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/6
    viYG5xJExU6',
    ],
];
```

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, `UserProfileFixture` 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'];
}
```

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()` and `yii\test\Fixture::unload()` methods.

13.2.2 Using Fixtures

If you are using CodeCeption[1] to test your code, you should consider using the `yii2-codeception` extension which has the 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()` 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.

[1] http://codeception.com/

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.

13.2.3 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.

13.2.4 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 for `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.

13.2.5 Summary

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

13.3 Managing Fixtures

// todo: this tutorial may be merged into test-fixture.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.

13.3.1 Fixtures format

Fixtures are objects with different methods and configurations, refer to official documentation[2] 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/6
    viYG5xJExU6',
    ],
];
```

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[3]. Above fixture example was auto-generated by `yii2-faker` extension, read more about it in these section. Fixture classes name should not be plural.

[2] https://github.com/yiisoft/yii2/blob/master/docs/guide/test-fixture.md

[3] https://github.com/yiisoft/yii2/blob/master/docs/guide/test-fixture.md

13.3.2 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.

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 `users` fixture
yii fixture/load User

// same as above, because default action of "fixture" command is "load"
yii fixture User

// load several fixtures. Note that there should not be any whitespace
    between ",", it should be one string.
yii fixture User,UserProfile

// load all fixtures
yii fixture/load all

// same as above
yii fixture all

// load fixtures, but for other database connection.
yii fixture User --db='customDbConnectionId'

// load fixtures, but search them in different namespace. By default
    namespace is: tests\unit\fixtures.
yii fixture User --namespace='alias\my\custom\namespace'

// load global fixture `some\name\space\CustomFixture` before other fixtures
     will be loaded.
// 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'
```

13.3.3 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. Note that there should not be any whitespace
    between ",", it should be one string.
```

```
yii fixture/unload User,UserProfile

// unload all fixtures
yii fixture/unload all
```

Same command options like: `db`, `namespace`, `globalFixtures` also can be applied to this command.

13.3.4 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',
        'db' => 'customDbConnectionId',
        'namespace' => 'myalias\some\custom\namespace',
        'globalFixtures' => [
            'some\name\space\Foo',
            'other\name\space\Bar'
        ],
    ],
]
```

13.3.5 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. These feature is done by Faker[4] library and `yii2-faker` extension. See extension guide[5] for more docs.

[4]https://github.com/fzaninotto/Faker
[5]https://github.com/yiisoft/yii2/tree/master/extensions/faker

Chapter 14

Extending Yii

14.1 Extending Yii

> Note: This section is under development.

The Yii framework was designed to be easily extendable. Additional features can be added to your project and then reused, either by yourself on other projects or by sharing your work as a formal Yii extension.

14.1.1 Code style

To be consistent with core Yii conventions, your extensions ought to adhere to certain coding styles:

- Use the core framework code style[1].
- Document classes, methods and properties using phpdoc[2]. - Extension classes should *not* be prefixed. Do not use the format `TbNavBar`, `EMyWidget`, etc.

  > Note that you can use Markdown within your code for documentation purposes. With Markdown, you can link to properties and methods using the following syntax: `[[name()]]`, `[[namespace\MyClass::name()]]`.

Namespace

Yii 2 relies upon namespaces to organize code. (Namespace support was added to PHP in version 5.3.) If you want to use namespaces within your extension,

[1]https://github.com/yiisoft/yii2/wiki/Core-framework-code-style
[2]http://www.phpdoc.org/

- Do not use `yiisoft` anywhere in your namespaces.
- Do not use `\yii`, `\yii2` or `\yiisoft` as root namespaces.
- Namespaces should use the syntax `vendorName\uniqueName`.

Choosing a unique namespace is important to prevent name collisions, and also results in faster autoloading of classes. Examples of unique, consistent namepacing are:

- `samdark\wiki`
- `samdark\debugger`
- `samdark\googlemap`

14.1.2 Distribution

Beyond the code itself, the entire extension distribution ought to have certain things.

There should be a `readme.md` file, written in English. This file should clearly describe what the extension does, its requirements, how to install it, and to use it. The README should be written using Markdown. If you want to provide translated README files, name them as `readme_ru.md` where `ru` is your language code (in this case, Russian).

It is a good idea to include some screenshots as part of the documentation, especially if your extension provides a widget.

It is recommended to host your extensions at Github[3].

Extensions should also be registered at Packagist[4] in order to be installable via Composer.

Composer package name

Choose your extension's package name wisely, as you shouldn't change the package name later on. (Changing the name leads to losing the Composer stats, and makes it impossible for people to install the package by the old name.)

If your extension was made specifically for Yii2 (i.e. cannot be used as a standalone PHP library) it is recommended to name it like the following:

```
yii2-my-extension-name-type
```

Where:

- `yii2-` is a prefix.

[3]`https://github.com`
[4]`https://packagist.org`

- The extension name is in all lowercase letters, with words separated by `-`.
- The `-type` postfix may be `widget`, `behavior`, `module` etc.

Dependencies

Some extensions you develop may have their own dependencies, such as relying upon other extensions or third-party libraries. When dependencies exist, you should require them in your extension's `composer.json` file. Be certain to also use appropriate version constraints, eg. `1.*`, `@stable` for requirements.

Finally, when your extension is released in a stable version, double-check that its requirements do not include `dev` packages that do not have a `stable` release. In other words, the stable release of your extension should only rely upon stable dependencies.

Versioning

As you maintain and upgrading your extension,

- Use the rules of semantic versioning[5].
- Use a consistent format for your repository tags, as they are treated as version strings by composer, eg. `0.2.4`, `0.2.5`,`0.3.0`,`1.0.0`.

composer.json

Yii2 uses Composer for installation, and extensions for Yii2 should as well. Towards that end,

- Use the type `yii2-extension` in `composer.json` file if your extension is Yii-specific.
- Do not use `yii` or `yii2` as the Composer vendor name.
- Do not use `yiisoft` in the Composer package name or the Composer vendor name.

If your extension classes reside directly in the repository root directory, you can use the PSR-4 autoloader in the following way in your `composer.json` file:

```
{
    "name": "myname/mywidget",
    "description": "My widget is a cool widget that does everything",
    "keywords": ["yii", "extension", "widget", "cool"],
    "homepage": "https://github.com/myname/yii2-mywidget-widget",
    "type": "yii2-extension",
```

[5]http://semver.org

```
    "license": "BSD-3-Clause",
    "authors": [
        {
            "name": "John Doe",
            "email": "doe@example.com"
        }
    ],
    "require": {
        "yiisoft/yii2": "*"
    },
    "autoload": {
        "psr-4": {
            "myname\\mywidget\\": ""
        }
    }
}
```

In the above, `myname/mywidget` is the package name that will be registered at Packagist[6]. It is common for the package name to match your Github repository name. Also, the `psr-4` autoloader is specified in the above, which maps the `myname\mywidget` namespace to the root directory where the classes reside.

More details on this syntax can be found in the Composer documentation[7].

Bootstrap with extension

Sometimes, you may want your extension to execute some code during the bootstrap stage of an application. For example, your extension may want to respond to the application's `beginRequest` event. You can ask the extension user to explicitly attach your event handler in the extension to the application's event. A better way, however, is to do all these automatically.

To achieve this goal, you can create a bootstrap class by implementing `yii\base\BootstrapInterface`.

```
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
        });
    }
}
```

[6]https://packagist.org
[7]http://getcomposer.org/doc/04-schema.md#autoload

You then list this bootstrap class in `composer.json` as follows,

```
{
    "extra": {
        "bootstrap": "myname\\mywidget\\MyBootstrapClass"
    }
}
```

When the extension is installed in an application, Yii will automatically hook up the bootstrap class and call its `bootstrap()` while initializing the application for every request.

14.1.3 Working with database

Extensions sometimes have to use their own database tables. In such a situation,

- If the extension creates or modifies the database schema, always use Yii migrations instead of SQL files or custom scripts.
- Migrations should be applicable to different database systems.
- Do not use Active Record models in your migrations.

14.1.4 Assets

- Register assets through bundles.

14.1.5 Events

TBD

14.1.6 i18n

- If extension outputs messages intended for end user these should be wrapped into `Yii::t()` in order to be translatable.
- Exceptions and other developer-oriented message should not be translated.
- Consider proving `config.php` for `yii message` command to simplify translation.

14.1.7 Testing your extension

- Consider adding unit tests for PHPUnit.

14.2 Helper Classes

> 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 forth. 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:

```
use yii\helpers\ArrayHelper;

$c = ArrayHelper::merge($a, $b);
```

14.2.1 Extending Helper Classes

To make helper classes easier to extend, Yii breaks each 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, never use the base class.

If you want to customize a helper, perform the following steps (using `ArrayHelper` as an example):

1. Name your class the same as the concrete class provided by Yii, including the namespace: `yii\helpers\ArrayHelper`

2. Extend your class from the base class: `class ArrayHelper extends \yii\helpers\BaseArrayHelper`.

3. In your class, override any method or property as needed, or add new methods or properties.

4. Tell your application to use your version of the helper class by including the following line of code in the bootstrap script:

```
Yii::$classMap['yii\helpers\ArrayHelper'] = 'path/to/ArrayHelper.php';
```

Step 4 above will instruct the Yii class autoloader to load your version of the helper class instead of the one included in the Yii distribution.

> Tip: You can use `Yii::$classMap` to replace ANY core Yii class with your own customized version, not just helper classes.

14.3 Using 3rd-Party Libraries

Note: This section is under development.

Yii is carefully designed so that third-party libraries can be easily integrated to further extend Yii's functionalities.

14.3.1 Using Packages Installed via Composer

Packages installed via Composer can be directly used in Yii without any special handling.

14.3.2 Using Downloaded Libraries

If a library has its own class autoloader, please follow its instruction on how to install the autoloader.

If a library does not have a class autoloader, you may face one of the following scenarios:

- The library requires specific PHP include path configuration.
- The library requires explicitly including one or several of its files.
- Neither of the above.

In the last scenario, the library is not written very well, but you can still do the following work to make it work with Yii:

- Identify which classes the library contains.
- List the classes and the corresponding file paths in `Yii::$classMap`.

For example, if none of the classes in a library is namespaced, you may register the classes with Yii like the following in the entry script after including `yii.php`:

```
Yii::$classMap['Class1'] = 'path/to/Class1.php';
Yii::$classMap['Class2'] = 'path/to/Class2.php';
// ...
```

14.3.3 Using Yii in 3rd-Party Systems

Yii can also be used as a self-contained library to support developing and enhancing existing 3rd-party systems, such as WordPress, Joomla, etc. To do so, include the following code in the bootstrap code of the 3rd-party system:

```
$yiiConfig = require(__DIR__ . '/../config/yii/web.php');
new yii\web\Application($yiiConfig); // No 'run()' invocation!
```

The above code is very similar to the bootstrap code used by a typical Yii application except one thing: it does not call the `run()` method after creating the Web application instance.

Now we can use most features offered by Yii when developing 3rd-party enhancements. For example, we can use `Yii::$app` to access the application instance; we can use the database features such as ActiveRecord; we can use the model and validation feature; and so on.

14.3.4 Using Yii2 with Yii1

Yii2 can be used along with Yii1 at the same project. Since Yii2 uses namespaced class names they will not conflict with any class from Yii1. However there is single class, which name is used both in Yii1 and Yii2, it named 'Yii'. In order to use both Yii1 and Yii2 you need to resolve this collision. To do so you need to define your own 'Yii' class, which will combine content of 'Yii' from 1.x and 'Yii' from 2.x.

When using composer you add the following to your composer.json in order to add both versions of yii to your project:

```
"require": {
    "yiisoft/yii": "*",
    "yiisoft/yii2": "*",
},
```

Start from defining your own descendant of `yii\BaseYii`:

```
$yii2path = '/path/to/yii2';
require($yii2path . '/BaseYii.php');

class Yii extends \yii\BaseYii
{
}

Yii::$classMap = include($yii2path . '/classes.php');
```

Now we have a class, which suites Yii2, but causes fatal errors for Yii1. So, first of all, we need to include `YiiBase` of Yii1 source code to our 'Yii' class definition file:

```
$yii2path = '/path/to/yii2';
require($yii2path . '/BaseYii.php'); // Yii 2.x
$yii1path = '/path/to/yii1';
require($yii1path . '/YiiBase.php'); // Yii 1.x

class Yii extends \yii\BaseYii
{
}

Yii::$classMap = include($yii2path . '/classes.php');
```

Using this, defines all necessary constants and autoloader of Yii1. Now we need to add all fields and methods from `YiiBase` of Yii1 to our 'Yii' class. Unfortunately, there is no way to do so but copy-paste:

```
$yii2path = '/path/to/yii2';
require($yii2path . '/BaseYii.php');
$yii1path = '/path/to/yii1';
require($yii1path . '/YiiBase.php');

class Yii extends \yii\BaseYii
{
    public static $classMap = [];
    public static $enableIncludePath = true;
    private static $_aliases = ['system'=>YII_PATH,'zii'=>YII_ZII_PATH];
    private static $_imports = [];
    private static $_includePaths;
    private static $_app;
    private static $_logger;

    public static function getVersion()
    {
        return '1.1.15-dev';
    }

    public static function createWebApplication($config=null)
    {
        return self::createApplication('CWebApplication',$config);
    }

    public static function app()
    {
        return self::$_app;
    }

    // Rest of \YiiBase internal code placed here
    ...
}

Yii::$classMap = include($yii2path . '/classes.php');
Yii::registerAutoloader(['Yii', 'autoload']); // Register Yii2 autoloader
    via Yii1
```

Note: while copying methods you should NOT copy method "autoload()"! Also you may avoid copying "log()", "trace()", "beginProfile()", "endProfile()" in case you want to use Yii2 logging instead of Yii1 one.

Now we have 'Yii' class, which suites both Yii 1.x and Yii 2.x. So bootstrap code used by your application will looks like following:

```
require(__DIR__ . '/../components/my/Yii.php'); // include created 'Yii'
    class

$yii2Config = require(__DIR__ . '/../config/yii2/web.php');
new yii\web\Application($yii2Config); // create Yii 2.x application
```

```
$yii1Config = require(__DIR__ . '/../config/yii1/main.php');
Yii::createWebApplication($yii1Config)->run(); // create Yii 1.x application
```

Then in any part of your program 'Yii::$app`refers to Yii 2.x application, while`Yii::app()' refers to Yii 1.x application:

```
echo get_class(Yii::app()); // outputs 'CWebApplication'
echo get_class(Yii::$app); // outputs 'yii\web\Application'
```

Error: not existing file: extend-embedding-in-others.md

Error: not existing file: extend-using-v1-v2.md

14.4 Composer

> Note: This section is under development.

Yii2 uses Composer as its dependency management tool. Composer is a PHP utility that can automatically handle the installation of needed libraries and extensions, thereby keeping those third-party resources up to date while absolving you of the need to manually manage the project's dependencies.

14.4.1 Installing Composer

In order to install Composer, check the official guide for your operating system:

- Linux[8]
- Windows[9]

All of the details can be found in the guide, but you'll either download Composer directly from `http://getcomposer.org/`, or run the following command:

```
curl -s http://getcomposer.org/installer | php
```

We strongly recommend a global composer installation.

14.4.2 Working with composer

The act of installing a Yii application with

```
composer.phar create-project --stability dev yiisoft/yii2-app-basic
```

creates a new root directory for your project along with the `composer.json` and `compoer.lock` file.

While the former lists the packages, which your application requires directly together with a version constraint, while the latter keeps track of all installed packages and their dependencies in a specific revision. Therefore the `composer.lock` file should also be committed to your version control system[10].

These two files are strongly linked to the two composer commands `update` and `install`. Usually, when working with your project, such as creating another copy for development or deployment, you will use

```
composer.phar install
```

to make sure you get exactly the same packages and versions as specified in `composer.lock`.

Only if want to intentionally update the packages in your project you should run

[8]http://getcomposer.org/doc/00-intro.md#installation-nix
[9]http://getcomposer.org/doc/00-intro.md#installation-windows
[10]https://getcomposer.org/doc/01-basic-usage.md#composer-lock-the-lock-file

```
composer.phar update
```

As an example, packages on `dev-master` will constantly get new updates when you run `update`, while running `install` won't, unless you've pulled an update of the `composer.lock` file.

There are several paramaters available to the above commands. Very commonly used ones are `--no-dev`, which would skip packages in the `require-dev` section and `--prefer-dist`, which downloads archives if available, instead of checking out repositories to your `vendor` folder.

> Composer commands must be executed within your Yii project's directory, where the `composer.json` file can be found. Depending upon your operating system and setup, you may need to provide paths to the PHP executable and to the `composer.phar` script.

14.4.3 Adding more packages to your project

To add two new packages to your project run the follwing command:

```
composer.phar require "michelf/php-markdown:>=1.3" "ezyang/htmlpurifier
    :>4.5.0"
```

This will resolve the dependencies and then update your `composer.json` file. The above example says that a version greater than or equal to 1.3 of Michaelf's PHP-Markdown package is required and version 4.5.0 or greater of Ezyang's HTMLPurifier.

For details of this syntax, see the official Composer documentation[11].

The full list of available Composer-supported PHP packages can be found at packagist[12]. You may also search packages interactively just by entering `composer.phar require`.

Manually editing your version constraints

You may also edit the `composer.json` file manually. Within the `require` section, you specify the name and version of each required package, same as with the command above.

```
{
    "require": {
        "michelf/php-markdown": ">=1.4",
        "ezyang/htmlpurifier": ">=4.6.0"
    }
}
```

Once you have edited the `composer.json`, you can invoke Composer to download the updated dependencies. Run

[11] `https://getcomposer.org/doc/01-basic-usage.md#package-versions`
[12] `http://packagist.org/`

```
composer.phar update michelf/php-markdown ezyang/htmlpurifier
```

afterwards.

> Depending on the package additional configuration may be required (eg. you have to register a module in the config), but autoloading of the classes should be handled by composer.

14.4.4 Using a specifc version of a package

Yii always comes with the latest version of a required library that it is compatible with, but allows you to use an older version if you need to.

A good example for this is jQuery which has dropped old IE browser support[13] in version 2.x. When installing Yii via composer the installed jQuery version will be the latest 2.x release. When you want to use jQuery 1.10 because of IE browser support you can adjust your composer.json by requiring a specific version of jQuery like this:

```
{
    "require": {
        ...
        "yiisoft/jquery": "1.10.*"
    }
}
```

14.4.5 FAQ

Getting "You must enable the openssl extension to download files via https"

If you're using WAMP check this answer at StackOverflow[14].

Getting "Failed to clone <URL here>, git was not found, check that it is installed and in your Path env."

Either install git or try adding `--prefer-dist` to the end of `install` or `update` command.

Should I Commit The Dependencies In My Vendor Directory?

Short answer: No. Long answer, see here[15].

[13]http://jquery.com/browser-support/
[14]http://stackoverflow.com/a/14265815/1106908
[15]https://getcomposer.org/doc/faqs/should-i-commit-the-dependencies-in-my-vendor-directory.md

14.4.6 See also

- Official Composer documentation[16].

[16]`http://getcomposer.org`

Chapter 15

Special Topics

15.1 Advanced application template

Note: This section is under development.

This template is for large projects developed in teams where the backend is divided from the frontend, application is deployed to multiple servers etc. This application template also goes a bit further regarding features and provides essential database, signup and password restore out of the box.

15.1.1 Installation

Install via Composer

If you do not have Composer[1], you may download it from `http://getcomposer.org/`[2] or run the following command on Linux/Unix/MacOS:
curl -sS `http://getcomposer.org/installer` | php
You can then install the application using the following command:
php composer.phar create-project –prefer-dist –stability=dev yiisoft/yii2-app-advanced /path/to/yii-application

15.1.2 Getting started

After you install the application, you have to conduct the following steps to initialize the installed application. You only need to do these once for all.

1. Execute the `init` command and select `dev` as environment.

   ```
   php /path/to/yii-application/init
   ```

   Otherwise, in production execute `init` in non-interactive mode.

   ```
   php /path/to/yii-application/init --env=Production overwrite=All
   ```

[1] http://getcomposer.org/
[2] http://getcomposer.org/

2. Create a new database and adjust the `components.db` configuration in `common/config/main-local.php` accordingly.
3. Apply migrations with console command `yii migrate`.
4. Set document roots of your web server:

- for frontend `/path/to/yii-application/frontend/web/` and using the URL `http://frontend/`
- for backend `/path/to/yii-application/backend/web/` and using the URL `http://backend/`

15.1.3 Directory structure

The root directory contains the following subdirectories:

- `backend` - backend web application.
- `common` - files common to all applications.
- `console` - console application.
- `environments` - environment configs.
- `frontend` - frontend web application.

Root directory contains a set of files.

- `.gitignore` contains a list of directories ignored by git version system. If you need something never get to your source code repository, add it there.
- `composer.json` - Composer config described in detail below.
- `init` - initialization script described in "Composer config described in detail below".
- `init.bat` - same for Windows.
- `LICENSE.md` - license info. Put your project license there. Especially when opensourcing.
- `README.md` - basic info about installing template. Consider replacing it with information about your project and its installation.
- `requirements.php` - Yii requirements checker.
- `yii` - console application bootstrap.
- `yii.bat` - same for Windows.

15.1.4 Predefined path aliases

- `@yii` - framework directory.
- `@app` - base path of currently running application.
- `@common` - common directory.
- `@frontend` - frontend web application directory.
- `@backend` - backend web application directory.
- `@console` - console directory.
- `@runtime` - runtime directory of currently running web application.
- `@vendor` - Composer vendor directory.
- `@web` - base URL of currently running web application.
- `@webroot` - web root directory of currently running web application.

The aliases specific to the directory structure of the advanced application (`@common`, `@frontend`, `@backend`, and `@console`) are defined in `common/config/aliases.php`.

15.1.5 Applications

There are three applications in advanced template: frontend, backend and console. Frontend is typically what is presented to end user, the project itself. Backend is admin panel, analytics and such functionality. Console is typically used for cron jobs and low-level server management. Also it's used during application deployment and handles migrations and assets.

There's also a `common` directory that contains files used by more than one application. For example, `User` model.

frontend and backend are both web applications and both contain the `web` directory. That's the webroot you should point your web server to.

Each application has its own namespace and alias corresponding to its name. Same applies to common directory.

15.1.6 Configuration and environments

There are multiple problems with a typical approach to configuration:

- Each team member has its own configuration options. Committing such config will affect other team members.
- Production database password and API keys should not end up in the repository.

- There are multiple server environments: development, testing, production. Each should have its own configuration.
- Defining all configuration options for each case is very repetitive and takes too much time to maintain.

In order to solve these issues Yii introduces a simple environments concept. Each environment is represented by a set of files under the `environments` directory. The `init` command is used to switch between these. What it really does is copy everything from the environment directory over to the root directory where all applications are.

Typically environment contains application bootstrap files such as `index.php` and config files suffixed with `-local.php`. These are added to `.gitignore` and never added to source code repository.

In order to avoid duplication configurations are overriding each other. For example, the frontend reads configuration in the following order:

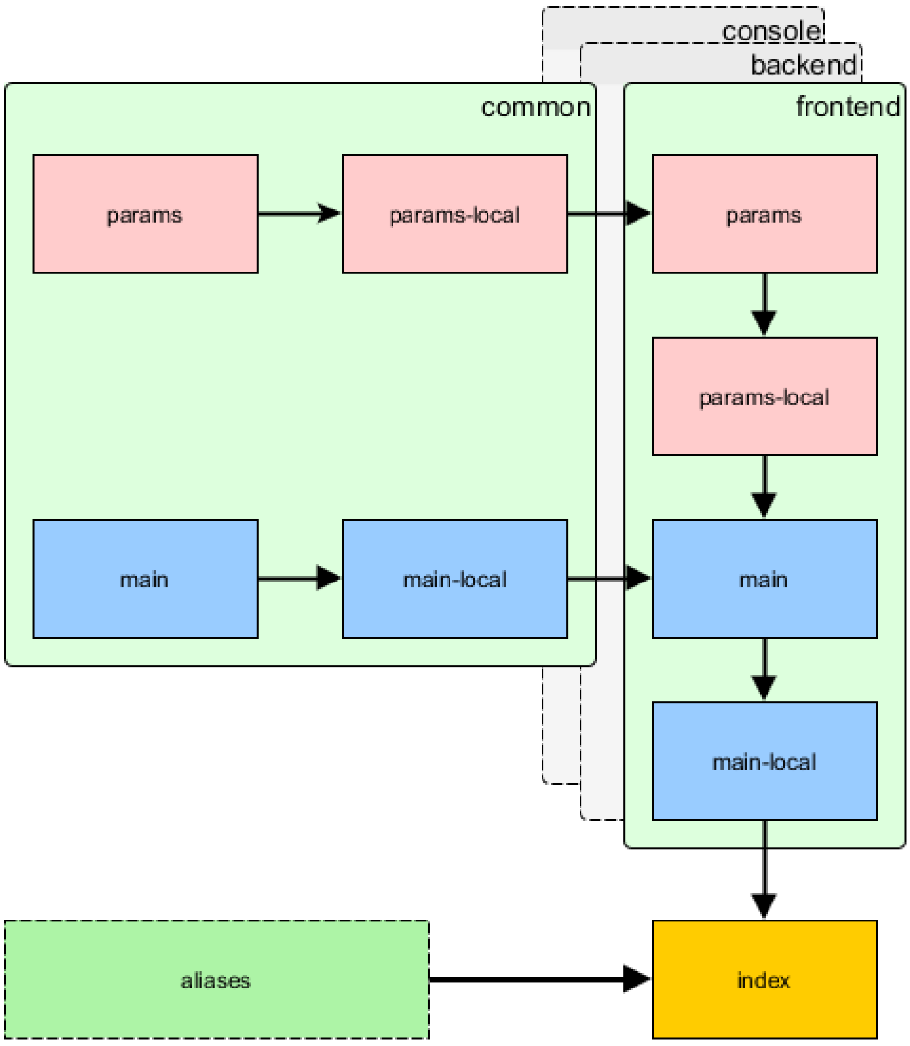
- `common/config/main.php`
- `common/config/main-local.php`
- `frontend/config/main.php`
- `frontend/config/main-local.php`

Parameters are read in the following order:

- `common/config/params.php`
- `common/config/params-local.php`
- `frontend/config/params.php`
- `frontend/config/params-local.php`

The later config file overrides the former.

Here's the full scheme:



15.1.7 Configuring Composer

After the application template is installed it's a good idea to adjust default `composer.json` that can be found in the root directory:

```
{
    "name": "yiisoft/yii2-app-advanced",
    "description": "Yii 2 Advanced Application Template",
    "keywords": ["yii", "framework", "advanced", "application template"],
    "homepage": "http://www.yiiframework.com/",
    "type": "project",
    "license": "BSD-3-Clause",
    "support": {
        "issues": "https://github.com/yiisoft/yii2/issues?state=open",
        "forum": "http://www.yiiframework.com/forum/",
        "wiki": "http://www.yiiframework.com/wiki/",
        "irc": "irc://irc.freenode.net/yii",
        "source": "https://github.com/yiisoft/yii2"
    },
    "minimum-stability": "dev",
    "require": {
        "php": ">=5.4.0",
        "yiisoft/yii2": "*",
        "yiisoft/yii2-swiftmailer": "*",
        "yiisoft/yii2-bootstrap": "*",
        "yiisoft/yii2-debug": "*",
        "yiisoft/yii2-gii": "*"
    },
    "scripts": {
        "post-create-project-cmd": [
            "yii\\composer\\Installer::setPermission"
        ]
    },
    "extra": {
        "writable": [
            "backend/runtime",
            "backend/web/assets",

            "console/runtime",
            "console/migrations",

            "frontend/runtime",
            "frontend/web/assets"
        ]
    }
}
```

First we're updating basic information. Change `name`, `description`, `keywords`, `homepage` and `support` to match your project.

Now the interesting part. You can add more packages your application needs to the `require` section. All these packages are coming from packagist.org[3] so feel free to browse the website for useful code.

After your `composer.json` is changed you can run `php composer.phar update --prefer-dist`, wait till packages are downloaded and installed and then just use them. Autoloading of classes will be handled automatically.

15.1.8 Creating links from backend to frontend

Often it's required to create links from the backend application to the frontend application. Since the frontend application may contain its own URL manager rules you need to duplicate that for the backend application by naming it differently:

```
return [
    'components' => [
```

[3] `https://packagist.org/`

```
        'urlManager' => [
            // here is your normal backend url manager config
        ],
        'urlManagerFrontend' => [
            // here is your frontend URL manager config
        ],

    ],
];
```

After it is done, you can get an URL pointing to frontend like the following:

```
echo Yii::$app->urlManagerFrontend->createUrl(...);
```

15.2 Creating your own Application structure

> Note: This section is under development.

While basic and advanced application templates are great for most of your needs you may want to create your own application template to start your projects with.

Application templates are repositories containing `composer.json` and registered as Composer packages so you can make any repository a package and it will be installable via `create-project` command.

Since it's a bit too much to start building your template from scratch it is better to use one of built-in templates as a base. Let's use basic template.

15.2.1 Clone basic template repository from git

```
git clone git@github.com:yiisoft/yii2-app-basic.git
```

And wait till it's downloaded. Since we don't need to push our changes back to Yii's repository we delete `.git` and all of its contents.

15.2.2 Modify files

Now we need to modify `composer.json`. Change `name`, `description`, `keywords`, `homepage`, `license`, `support` to match your new template. Adjust `require`, `require-dev`, `suggest` and the rest of the options.

> **Note**: In `composer.json` file `writable` under `extra` is functionality added by Yii that allows you to specify per file permissions to set after an application is created using the template.

Next actually modify the structure of the future application as you like and update readme.

15.2.3 Make a package

Create git repository and push your files there. If you're going to make it open source github is the best way to host it. If it should remain private, any git repository would do.

Then you need to register your package. For public templates it should be registered at packagist[4]. For private ones it is a bit more tricky but well defined in Composer documentation[5].

15.2.4 Use it

That's it. Now you can create projects using a template:

```
php composer.phar create-project --prefer-dist --stability=dev mysoft/yii2-
    app-coolone new-project
```

15.3 Console applications

> Note: This section is under development.

Yii has full featured support for console applications, whose structure is very similar to a Yii web application. A console application 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.

15.3.1 Usage

You execute a console controller action using the following syntax:

```
yii <route> [--option1=value1 --option2=value2 ... argument1 argument2 ...]
```

For example, the `yii\console\controllers\MigrateController::actionCreate()` with `yii\console\controllers\MigrateController::$migrationTable` set can be called from command line like so:

```
yii migrate/create --migrationTable=my_migration
```

In the above `yii` is the console application entry script described below.

15.3.2 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. The contents of the console application entry script contains code like the following:

[4]https://packagist.org/
[5]https://getcomposer.org/doc/05-repositories.md#hosting-your-own

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 */

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

// fcgi doesn't have STDIN and STDOUT defined by default
defined('STDIN') or define('STDIN', fopen('php://stdin', 'r'));
defined('STDOUT') or define('STDOUT', fopen('php://stdout', 'w'));

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 `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 to provide a more developer-friendly environment.

15.3.3 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 the 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" application template.

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 do 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 ...
```

15.3.4 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()` 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.

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 follow examples show 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 occured!\n";
        return 1;
    }
    // do something
    return 0;
}
```

15.4 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.

15.4.1 yii\validators\BooleanValidator

```
[
    // 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 property as false.

15.4.2 yii\captcha\CaptchaValidator

```
[
    ['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 widget.

- `caseSensitive`: whether the comparison of the verification code is case sensitive. Defaults to false.
- `captchaAction`: the route corresponding to the yii\captcha\CaptchaAction 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.

15.4.3 `yii\validators\CompareValidator`

```
[
    // validates if the value of "password" attribute equals to that of "
    password_repeat"
    ['password', 'compare'],

    // validates if age is greater than or equal to 30
    ['age', 'compare', 'compareValue' => 30, 'operator' => '>='],
]
```

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.

15.4.4 yii\validators\DateValidator

```
[
    [['from', 'to'], 'date'],
]
```

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 and store it in an attribute specified via `yii\validators\DateValidator::timestampAttribute`.

- `format`: the date/time format that the value being validated should be in. Please refer to the PHP manual about date_create_from_format()[6] for details about specifying the format string. The default value is `'Y-m-d'`.
- `timestampAttribute`: the name of the attribute to which this validator may assign the UNIX timestamp converted from the input date/time.

15.4.5 yii\validators\DefaultValueValidator

```
[
    // 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 empty
    [['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.

[6] http://www.php.net/manual/en/datetime.createfromformat.php

15.4.6 yii\validators\NumberValidator

```
[
    // 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.

15.4.7 yii\validators\EmailValidator

```
[
    // 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.

15.4.8 yii\validators\ExistValidator

```
[
    // 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. 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 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` 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.

15.4.9 `yii\validators\FileValidator`

```
[
    // checks if "primaryImage" is an uploaded image file in PNG, JPG or GIF
     format.
    // the file size must be less than 1MB
    ['primaryImage', 'file', 'types' => ['png', 'jpg', 'gif'], 'maxSize' =>
    1024*1024*1024],
]
```

This validator checks if the input is a valid uploaded file.

- `types`: 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.

- `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.

`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.

15.4.10 `yii\validators\FilterValidator`

```
[
    // 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 trim validator.

15.4.11 `yii\validators\ImageValidator`

```
[
    // checks if "primaryImage" is a valid image with proper size
    ['primaryImage', 'image', 'types' => '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.
- `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"). Mime type names are case-insensitive. Defaults to null, meaning all MIME types are allowed.

15.4.12 `yii\validators\RangeValidator`

```
[
    // 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.

15.4.13 `yii\validators\NumberValidator`

```
[
    // 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.

15.4.14 `yii\validators\RegularExpressionValidator`

```
[
    // 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.

15.4.15 `yii\validators\NumberValidator`

```
[
    // checks if "salary" is a number
    ['salary', 'number'],
]
```

This validator checks if the input value is a number. It is equivalent to the [double](#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.

15.4.16 `yii\validators\RequiredValidator`

```
[
    // 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.

15.4.17 `yii\validators\SafeValidator`

```
[
    // 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.

15.4.18 `yii\validators\StringValidator`

```
[
    // 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

Validates that the attribute value is of 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` value which defaults to `UTF-8`.

15.4.19 `yii\validators\FilterValidator`

```
[
    // 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.

15.4.20 `yii\validators\UniqueValidator`

```
[
    // 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 the 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 value)
    ['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` object that you can modify in the function.

15.4.21 `yii\validators\UrlValidator`

```
[
    // 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.

15.5 Internationalization

> Note: This section is under development.

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.

15.5.1 Locale and Language

There are two languages defined in Yii application: `yii\base\Application::$sourceLanguage` and `yii\base\Application::$language`.

Source language is the language original application messages are written in such as:

```
echo \Yii::t('app', 'I am a message!');
```

> **Tip**: Default is English and it's not recommended to change it. The reason is that it's easier to find people translating from English to any language than from non-English to non-English.

Target language is what's currently used. It's defined in application configuration like the following:

```
// ...
return [
    'id' => 'applicationID',
    'basePath' => dirname(__DIR__),
    'language' => 'ru-RU' // <- here!
```

Later you can easily change it in runtime:

```
\Yii::$app->language = 'zh-CN';
```

Format is `ll-CC` where `ll` is two- or three-letter lowercase code for a language according to ISO-639[7] and `CC` is country code according to ISO-3166[8].

If there's no translation for `ru-RU` Yii will try `ru` as well before failing.

> **Note**: you can further customize details specifying language as documented in ICU project[9].

[7] http://www.loc.gov/standards/iso639-2/

[8] http://www.iso.org/iso/en/prods-services/iso3166ma/02iso-3166-code-lists/list-en1.html

[9] http://userguide.icu-project.org/locale#TOC-The-Locale-Concept

15.5.2 Message translation

Basics

Yii basic message translation in its basic variant works without additional PHP extension. What it does is finding a translation of the message from source language into target language. Message itself is specified as the second `\Yii::t` method parameter:

```
echo \Yii::t('app', 'This is a string to translate!');
```

Yii tries to load appropriate translation from one of the message sources defined via `i18n` component configuration:

```
'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 `app*` is a pattern that specifies which categories are handled by the message source. In this case we're handling everything that begins with `app`. You can also specify default translation, for more info see this example.

`class` defines which message source is used. The following message sources are available:

- PhpMessageSource that uses PHP files.
- GettextMessageSource that uses GNU Gettext MO or PO files.
- DbMessageSource that uses database.

`basePath` defines where to store messages for the currently used message source. In this case it's `messages` directory in your application directory. In case of using database this option should be skipped.

`sourceLanguage` defines which language is used in `\Yii::t` second argument. If not specified, application's source language is used.

`fileMap` specifies how message categories specified in the first argument of `\Yii::t()` are mapped to files when `PhpMessageSource` is used. In the example we're defining two categories `app` and `app/error`.

Instead of configuring `fileMap` you can rely on convention which is `BasePath/messages/LanguageID/CategoryName.php`.

Named placeholders You can add parameters to a translation message that will be substituted with the corresponding value after translation. The format for this is to use curly brackets around the parameter name as you can see in the following example:

```
$username = 'Alexander';
echo \Yii::t('app', 'Hello, {username}!', [
    'username' => $username,
]);
```

Note that the parameter assignment is without the brackets.

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0}', $sum);
```

> **Tip**: Try keep message strings meaningful and avoid using too many positional parameters. Remember that translator has source string only so it should be obvious about what will replace each placeholder.

Advanced placeholder formatting

In order to use advanced features you need to install and enable intl[10] PHP extension. After installing and enabling it you will be able to use extended syntax for placeholders. Either short form `{placeholderName, argumentType }` that means default setting or full form `{placeholderName, argumentType, argumentStyle}` that allows you to specify formatting style.

Full reference is available at ICU website[11] but since it's a bit cryptic we have our own reference below.

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number}', $sum);
```

You can specify one of the built-in styles (`integer`, `currency`, `percent`):

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number, currency}', $sum);
```

Or specify custom pattern:

```
$sum = 42;
echo \Yii::t('app', 'Balance: {0, number, ,000,000000}', $sum);
```

Formatting reference[12].

[10]http://www.php.net/manual/en/intro.intl.php
[11]http://icu-project.org/apiref/icu4c/classMessageFormat.html
[12]http://icu-project.org/apiref/icu4c/classicu_1_1DecimalFormat.html

```
echo \Yii::t('app', 'Today is {0, date}', time());
```

Built in formats (`short`, `medium`, `long`, `full`):

```
echo \Yii::t('app', 'Today is {0, date, short}', time());
```

Custom pattern:

```
echo \Yii::t('app', 'Today is {0, date, YYYY-MM-dd}', time());
```

Formatting reference[13].

```
echo \Yii::t('app', 'It is {0, time}', time());
```

Built in formats (`short`, `medium`, `long`, `full`):

```
echo \Yii::t('app', 'It is {0, time, short}', time());
```

Custom pattern:

```
echo \Yii::t('app', 'It is {0, date, HH:mm}', time());
```

Formatting reference[14].

```
echo \Yii::t('app', '{n,number} is spelled as {n, spellout}', ['n' => 42]);
```

```
echo \Yii::t('app', 'You are {n, ordinal} visitor here!', ['n' => 42]);
```

Will produce "You are 42nd visitor here!".

```
echo \Yii::t('app', 'You are here for {n, duration} already!', ['n' => 47]);
```

Will produce "You are here for 47 sec. already!".

Plurals Different languages have different ways to inflect plurals. Some rules are very complex so it's very handy that this functionality is provided without the need to specify inflection rule. Instead it only requires your input of inflected word in certain situations.

```
echo \Yii::t('app', 'There {n, plural, =0{are no cats} =1{is one cat} other{
    are # cats}}!', ['n' => 0]);
```

[13] `http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html`
[14] `http://icu-project.org/apiref/icu4c/classicu_1_1SimpleDateFormat.html`

Will give us "There are no cats!".

In the plural rule arguments above `=0` means exactly zero, `=1` stands for exactly one `other` is for any other number. `#` is replaced with the `n` argument value. It's not that simple for languages other than English. Here's an example for Russian:

```
Здесь
 {n, plural, котов=0{ нет} есть=1{ один кот} one{# кот} few{# кота} many{#
    котов} other{# кота}}!
```

In the above it worth mentioning that `=1` matches exactly `n = 1` while `one` matches `21` or `101`.

Note that if you are using placeholder twice and one time it's used as plural another one should be used as number else you'll get "Inconsistent types declared for an argument: U_ARGUMENT_TYPE_MISMATCH" error:

```
Total {count, number} {count, plural, one{item} other{items}}.
```

To learn which inflection forms you should specify for your language you can referrer to rules reference at unicode.org[15].

Selections You can select phrases based on keywords. The pattern in this case specifies how to map keywords to phrases and provides a default phrase.

```
echo \Yii::t('app', '{name} is {gender} and {gender, select, female{she}
    male{he} other{it}} loves Yii!', [
    'name' => 'Snoopy',
    'gender' => 'dog',
]);
```

Will produce "Snoopy is dog and it loves Yii!".

In the expression `female` and `male` are possible values. `other` handler values that do not match. Strings inside brackets are sub-expressions so could be just a string or a string with more placeholders.

Specifying default translation

You can specify default translation that will be used as a fallback for categories that don't match any other translation. This translation should be marked with `*`. In order to do it add the following to the config file (for the `yii2-basic` application it will be `web.php`):

```
//configure i18n component

'i18n' => [
    'translations' => [
        '*' => [
```

[15]http://unicode.org/repos/cldr-tmp/trunk/diff/supplemental/language_plural_rules.html

```
            'class' => 'yii\i18n\PhpMessageSource'
        ],
    ],
],
```

Now you can use categories without configuring each one that is similar to Yii 1.1 behavior. Messages for the category will be loaded from a file under default translation `basePath` that is `@app/messages`:

```
echo Yii::t('not_specified_category', 'message from unspecified category');
```

Message will be loaded from `@app/messages/<LanguageCode>/not_specified_category.php`.

Translating module messages

If you want to translate messages for a module and avoid using a single translation file for all messages, you can make it like the following:

```
<?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 convention of category mapping to the same named file and use `Module::t('validation', 'your custom validation message')` or `Module::t('form', 'some form label')` directly.

Translating widgets messages

Same rules can be applied for widgets too, for example:

```
<?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 convention of category mapping to the same named file and use `Menu::t('messages', 'new messages {messages}', ['{messages}' => 10])` directly.

> **Note**: For widgets you also can use i18n views, same rules as for controllers are applied to them too.

Translating framework messages

Sometimes you want to correct default framework message translation for your application. In order to do so configure `i18n` component like the following:

```
'components' => [
    'i18n' => [
        'translations' => [
            'yii' => [
                'class' => 'yii\i18n\PhpMessageSource',
                'sourceLanguage' => 'en-US',
                'basePath' => '/path/to/my/message/files'
            ],
        ],
    ],
],
```

Now you can place your adjusted translations to `/path/to/my/message/files`.

Handling missing translations

If the translation is missing at the source, Yii displays the requested message content. Such behavior 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 requested translation is missing at the source. This can be achieved via 'missingTranslation' event of the `yii\i18n\MessageSource`.

For example: lets mark all missing translations with something notable, so they can be easily found at the page. First we need to setup event handler, this can be done via configuration:

```
'components' => [
    // ...
    'i18n' => [
        'translations' => [
            'app*' => [
                'class' => 'yii\i18n\PhpMessageSource',
                'fileMap' => [
                    'app' => 'app.php',
                    'app/error' => 'error.php',
                ],
                'on missingTranslation' => ['TranslationEventHandler', '
    handleMissingTranslation']
            ],
        ],
    ],
],
```

Now we need to implement own handler:

```
use yii\i18n\MissingTranslationEvent;

class TranslationEventHandler
{
    public static function(MissingTranslationEvent $event) {
        $event->translatedMessage = "@MISSING: {$event->category}.{$event->
    message} FOR LANGUAGE {$event->language} @";
    }
}
```

If `yii\i18n\MissingTranslationEvent::translatedMessage` is set by event handler it will be displayed as translation result.

> Attention: each message source handles its missing translations separately. If you are using several message sources and wish them treat missing translation in the same way, you should assign corresponding event handler to each of them.

15.5.3 Views

You can use i18n in your views to provide support for different languages. For example, if you have view `views/site/index.php` and you want to create special case for russian language, you create `ru-RU` folder under the view path of current controller/widget and put there file for russian language as follows `views/site/ru-RU/index.php`.

> **Note**: If language is specified as `en-US` and there are no corresponding views, Yii will try views under `en` before using original ones.

15.5.4 i18n formatter

i18n formatter component is the localized version of formatter that supports formatting of date, time and numbers based on current locale. In order to use it you need to configure formatter application component as follows:

```
return [
    // ...
    'components' => [
        'formatter' => [
            'class' => 'yii\i18n\Formatter',
        ],
    ],
];
```

After configuring the component can be accessed as `Yii::$app->formatter`.

Note that in order to use i18n formatter you need to install and enable intl[16] PHP extension.

In order to learn about formatter methods refer to its API documentation: `yii\i18n\Formatter`.

15.6 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[17] official extension.

15.6.1 Configuration

Mail component configuration depends on the extension you have chosen. In general your application configuration should look like:

```
return [
    //....
    'components' => [
        'mail' => [
            'class' => 'yii\swiftmailer\Mailer',
        ],
    ],
];
```

15.6.2 Basic usage

Once 'mail' component is configured, you can use the following code to send an email message:

```
Yii::$app->mail->compose()
    ->setFrom('from@domain.com')
    ->setTo('to@domain.com')
    ->setSubject('Message subject')
```

[16]http://www.php.net/manual/en/intro.intl.php

[17]https://github.com/yiisoft/yii2/tree/master/extensions/swiftmailer

```
    ->setTextBody('Plain text content')
    ->setHtmlBody('<b>HTML content</b>')
    ->send();
```

In above example 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->mail->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 'mail' extension comes in 2 major classes: 'Mailer' and 'Message'. 'Mailer' always knows the class name and specific of the 'Message'. Do not attempt ot 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->mail->compose()
        // ...
        ->setTo($user->email);
}
Yii::$app->mail->sendMultiple($messages);
```

Some particular mail extensions may benefit from this approach, using single network message etc.

15.6.3 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 \yii\web\View $this view component instance
 * @var \yii\mail\BaseMessage $message instance of newly created mail
    message
 */
```

```
?>
<h2>This message allows you to visit out 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->mail->compose('home-link') // message body becomes a view
    rendering result 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::$app->mail->compose('greetings', [
    'user' => Yii::$app->user->identity,
    'advertisement' => $adContent,
]);
```

You can specify different view files for HTML and plain text message contents:

```
Yii::$app->mail->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 an 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 \yii\web\View $this view component instance
 * @var string $content 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() ?>
```

15.6.4 File attachment

You can add attachments to message using methods `attach()` and `attachContent()`:

```
$message = Yii::$app->mail->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', '
    contentType' => 'text/plain']);
```

15.6.5 Embed 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->mail->compose('embed-email', ['imageFileName' => '/path/to/image.
    jpg'])
    // ...
    ->send();
```

Then inside view file you can use following code:

```
<img src="<?= $message->embed($imageFileName); ?>">
```

15.6.6 Testing and debugging

Developer often a to check, what actual emails are sent by 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 messages to the file or send them to actual recipients, but can not do both simultaneously.

Mail message file can be opened by regular text file editor, so you can browse actual message headers, content and so on. This mechanism amy prove itself, while debugging application or running unit test.

> Note: mail message file content is composed via `\yii\mail\MessageInterface::toString()`, so it depends on actual mail extension you are using in your application.

15.6.7 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 a base classes for your solution. These classes already contains 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 abstract methods to build you solution.

15.7 Performance Tuning

> Note: This section is under development.

The performance of your web application is based upon two parts. First is the framework performance and the second is the application itself. Yii has a pretty low performance impact on your application out of the box and can be fine-tuned further for production environment. As for the application, we'll provide some of the best practices along with examples on how to apply them to Yii.

15.7.1 Preparing environment

A well configured environment to run PHP application really matters. In order to get maximum performance:

- Always use latest stable PHP version. Each major release brings significant performance improvements and reduced memory usage.
- Use APC[18] for PHP 5.4 and less or Opcache[19] for PHP 5.5 and more. It gives a very good performance boost.

[18] `http://ru2.php.net/apc`
[19] `http://php.net/opcache`

15.7.2 Preparing framework for production

Disabling Debug Mode

First thing you should do before deploying your application to production environment is to disable debug mode. A Yii application runs in debug mode if the constant `YII_DEBUG` is defined as `true` in `index.php` so to disable debug the following should be in your `index.php`:

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

Debug mode is very useful during development stage, but it would impact performance because some components cause extra burden in debug mode. For example, the message logger may record additional debug information for every message being logged.

Enabling PHP opcode cache

Enabling the PHP opcode cache improves any PHP application performance and lowers memory usage significantly. Yii is no exception. It was tested with both PHP 5.5 OPcache[20] and APC PHP extension[21]. Both cache and optimize PHP intermediate code and avoid the time spent in parsing PHP scripts for every incoming request.

Turning on ActiveRecord database schema caching

If the application is using Active Record, we should turn on the schema caching to save the time of parsing database schema. This can be done by setting the `Connection::enableSchemaCache` property to be `true` via application configuration `protected/config/main.php`:

```
return [
    // ...
    'components' => [
        // ...
        '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. Default is 'cache'.
            //'schemaCache' => 'cache',
        ],
```

[20]http://php.net/manual/en/book.opcache.php
[21]http://php.net/manual/en/book.apc.php

```
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
    ],
];
```

Note that `cache` application component should be configured.

Combining and Minimizing Assets

It is possible to combine and minimize assets, typically JavaScript and CSS, in order to slightly improve page load time and therefore deliver better experience for end user of your application.

In order to learn how it can be achieved, refer to assets guide section.

Using better storage for sessions

By default PHP uses files to handle sessions. It is OK for development and small projects but when it comes to handling concurrent requests it's better to switch to another storage such as database. You can do so by configuring your application via `protected/config/main.php`:

```
return [
    // ...
    'components' => [
        'session' => [
            'class' => 'yii\web\DbSession',

            // Set the following if want to use DB component other than
            // default 'db'.
            // 'db' => 'mydb',

            // To override default session table set the following
            // 'sessionTable' => 'my_session',
        ],
    ],
];
```

You can use `CacheSession` to store sessions using cache. Note that some cache storage such as memcached has no guarantee that session data will not be lost leading to unexpected logouts.

If you have Redis[22] on your server, it's highly recommended as session storage.

15.7.3 Improving application

Using Serverside Caching Techniques

As described in the Caching section, Yii provides several caching solutions that may improve the performance of a Web application significantly. If the

[22] `http://redis.io/`

generation of some data takes long time, we can use the data caching approach to reduce the data generation frequency; If a portion of page remains relatively static, we can use the fragment caching approach to reduce its rendering frequency; If a whole page remains relative static, we can use the page caching approach to save the rendering cost for the whole page.

Leveraging HTTP to save processing time and bandwidth

Leveraging HTTP caching saves both processing time, bandwidth and resources significantly. It can be implemented by sending either `ETag` or `Last-Modified` header in your application response. If browser is implemented according to HTTP specification (most browsers are), content will be fetched only if it is different from what it was prevously.

Forming proper headers is time consuming task so Yii provides a shortcut in form of controller filter `yii\filters\HttpCache`. Using it is very easy. In a controller you need to implement `behaviors` method like the following:

```
public function behaviors()
{
    return [
        'httpCache' => [
            'class' => \yii\filters\HttpCache::className(),
            'only' => ['list'],
            'lastModified' => function ($action, $params) {
                $q = new Query();
                return strtotime($q->from('users')->max('updated_timestamp')
    );
            },
            // 'etagSeed' => function ($action, $params) {
                // return // generate etag seed here
            //}
        ],
    ];
}
```

In the code above one can use either `etagSeed` or `lastModified`. Implementing both isn't necessary. The goal is to determine if content was modified in a way that is cheaper than fetching and rendering that content. `lastModified` should return unix timestamp of the last content modification while `etagSeed` should return a string that is then used to generate `ETag` header value.

Database Optimization

Fetching data from database is often the main performance bottleneck in a Web application. Although using caching may alleviate the performance hit, it does not fully solve the problem. When the database contains enormous data and the cached data is invalid, fetching the latest data could be prohibitively expensive without proper database and query design.

Design index wisely in a database. Indexing can make SELECT queries much faster, but it may slow down INSERT, UPDATE or DELETE queries.

For complex queries, it is recommended to create a database view for it instead of issuing the queries inside the PHP code and asking DBMS to parse them repetitively.

Do not overuse Active Record. Although Active Record is good at modeling data in an OOP fashion, it actually degrades performance due to the fact that it needs to create one or several objects to represent each row of query result. For data intensive applications, using DAO or database APIs at lower level could be a better choice.

Last but not least, use `LIMIT` in your `SELECT` queries. This avoids fetching overwhelming data from database and exhausting the memory allocated to PHP.

Using asArray

A good way to save memory and processing time on read-only pages is to use ActiveRecord's `asArray` method.

```
class PostController extends Controller
{
    public function actionIndex()
    {
        $posts = Post::find()->orderBy('id DESC')->limit(100)->asArray()->
    all();
        return $this->render('index', ['posts' => $posts]);
    }
}
```

In the view you should access fields of each individual record from `$posts` as array:

```
foreach ($posts as $post) {
    echo $post['title']."<br>";
}
```

Note that you can use array notation even if `asArray` wasn't specified and you're working with AR objects.

Processing data in background

In order to respond to user requests faster you can process heavy parts of the request later if there's no need for immediate response.

- Cron jobs + console.
- queues + handlers.

TBD

If nothing helps

If nothing helps, never assume what may fix performance problem. Always profile your code instead before changing anything. The following tools may be helpful:

- Yii debug toolbar and debugger
- XDebug profiler[23]
- XHProf[24]

[23]`http://xdebug.org/docs/profiler`
[24]`http://www.php.net/manual/en/book.xhprof.php`

Error: not existing file: tutorial-shared-hosting.md

15.8 Using template engines

Note: This section is under development.

By default, Yii uses PHP as its template language, but you can configure Yii to support other rendering engines, such as Twig[25] or Smarty[26].

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',
                    //'options' => [], /*  Array of twig options */
                    'globals' => ['html' => '\yii\helpers\Html'],
                ],
                // ...
            ],
        ],
    ],
]
```

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": "*",
"yiisoft/yii2-twig": "*",
```

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.

15.8.1 Twig

To use Twig, you need to create templates in files that have the `.twig` extension (or use another file extension but configure the component accordingly). Unlike standard view files, when using Twig you must include the extension in your `$this->render()` controller call:

```
return $this->render('renderer.twig', ['username' => 'Alex']);
```

[25] `http://twig.sensiolabs.org/`
[26] `http://www.smarty.net/`

Template syntax

The best resource to learn Twig basics is its official documentation you can find at twig.sensiolabs.org[27]. Additionally there are Yii-specific addtions described below.

Method and function calls If you need result you can call a method or a function using the following syntax:

```
{% set result = my_function({'a' : 'b'}) %}
{% set result = myObject.my_function({'a' : 'b'}) %}
```

If you need to echo result instead of assigning it to a variable:

```
{{ my_function({'a' : 'b'}) }}
{{ myObject.my_function({'a' : 'b'}) }}
```

In case you don't need result you shoud use `void` wrapper:

```
{{ void(my_function({'a' : 'b'})) }}
{{ void(myObject.my_function({'a' : 'b'})} }}
```

Forms There are two form helper functions `form_begin` and `form_end` to make using forms more convenient:

```
{% set form = form_begin({
    'id' : 'login-form',
    'options' : {'class' : 'form-horizontal'},
}) %}
    {{ form.field(model, 'username') | raw }}
    {{ form.field(model, 'password').passwordInput() | raw }}

    <div class="form-group">
        <input type="submit" value="Login" class="btn btn-primary" />
    </div>
{{ form_end() }}
```

URLs There are two functions you can use for building URLs:

```
<a href="{{ path('blog/view', {'alias' : post.alias}) }}">{{ post.title }}</
    a>
<a href="{{ url('blog/view', {'alias' : post.alias}) }}">{{ post.title }}</a
    >
```

`path` generates relative URL while `url` generates absolute one. Internally both are using `\yii\helpers\Url`.

[27]http://twig.sensiolabs.org/documentation

Additional variables Within Twig templates the following variables are always defined:

- `app`, which equates to `\Yii::$app`
- `this`, which equates to the current `View` object

Additional configuration

Yii Twig extension allows you to define your own syntax and bring regular helper classes into templates. Let's review configuration options.

Globals You can add global helpers or values via the application configuration's `globals` variable. You can define both Yii helpers and your own variables there:

```
'globals' => [
    'html' => '\yii\helpers\Html',
    'name' => 'Carsten',
    'GridView' => '\yii\grid\GridView',
],
```

Once configured, in your template you can use the globals in the following way:

```
Hello, {{name}}! {{ html.a('Please login', 'site/login') | raw }}.

{{ GridView.widget({'dataProvider' : provider}) | raw }}
```

Functions You can define additional functions like the following:

```
'functions' => [
    'rot13' => 'str_rot13',
    'truncate' => '\yii\helpers\StringHelper::truncate',
],
```

In template they could be used like the following:

```
`{{ rot13('test') }}`
`{{ truncate(post.text, 100) }}`
```

Filters Additional filters may be added via the application configuration's `filters` option:

```
'filters' => [
    'jsonEncode' => '\yii\helpers\Json::encode',
],
```

Then in the template you can apply filter using the following syntax:

```
{{ model|jsonEncode }}
```

15.8.2 Smarty

To use Smarty, you need to create templates in files that have the `.tpl` extension (or use another file extension but configure the component accordingly). Unlike standard view files, when using Smarty you must include the extension in your `$this->render()` or `$this->renderPartial()` controller calls:

```
return $this->render('renderer.tpl', ['username' => 'Alex']);
```

Additional functions

Yii adds the following construct to the standard Smarty syntax:

```
<a href="{path route='blog/view' alias=$post.alias}">{$post.title}</a>
```

Internally, the `path()` function calls Yii's `Url::to()` method.

Additional variables

Within Smarty templates, you can also make use of these variables:

- `$app`, which equates to `\Yii::$app`
- `$this`, which equates to the current `View` object

Chapter 16

Widgets

16.1 Bootstrap Widgets

> Note: This section is under development.

Out of the box, Yii includes support for the Bootstrap 3[1] markup and components framework (also known as "Twitter Bootstrap"). Bootstrap is an excellent, responsive framework that can greatly speed up the client-side of your development process.

The core of Bootstrap is represented by two parts:

- CSS basics, such as a grid layout system, typography, helper classes, and responsive utilities.
- Ready to use components, such as forms, menus, pagination, modal boxes, tabs etc.

16.1.1 Basics

Yii doesn't wrap the bootstrap basics into PHP code since HTML is very simple by itself in this case. You can find details about using the basics at bootstrap documentation website[2]. Still Yii provides a convenient way to include bootstrap assets in your pages with a single line added to `AppAsset.php` located in your `@app/assets` directory:

```
public $depends = [
    'yii\web\YiiAsset',
    'yii\bootstrap\BootstrapAsset', // this line
    // 'yii\bootstrap\BootstrapThemeAsset' // uncomment to apply bootstrap 2
     style to bootstrap 3
];
```

Using bootstrap through Yii asset manager allows you to minimize its resources and combine with your own resources when needed.

[1] `http://getbootstrap.com/`
[2] `http://getbootstrap.com/css/`

16.1.2 Yii widgets

Most complex bootstrap components are wrapped into Yii widgets to allow more robust syntax and integrate with framework features. All widgets belong to `\yii\bootstrap` namespace:

- `yii\bootstrap\ActiveForm`
- `yii\bootstrap\Alert`
- `yii\bootstrap\Button`
- `yii\bootstrap\ButtonDropdown`
- `yii\bootstrap\ButtonGroup`
- `yii\bootstrap\Carousel`
- `yii\bootstrap\Collapse`
- `yii\bootstrap\Dropdown`
- `yii\bootstrap\Modal`
- `yii\bootstrap\Nav`
- `yii\bootstrap\NavBar`
- `yii\bootstrap\Progress`
- `yii\bootstrap\Tabs`

16.1.3 Using the .less files of Bootstrap directly

If you want to include the Bootstrap css directly in your less files[3] you may need to disable the original bootstrap css files to be loaded. You can do this by setting the css property of the `yii\bootstrap\BootstrapAsset` to be empty. For this you need to configure the `assetManager` application component as follows:

```
'assetManager' => [
    'bundles' => [
        'yii\bootstrap\BootstrapAsset' => [
            'css' => [],
        ]
    ]
]
```

[3]http://getbootstrap.com/getting-started/#customizing

Error: not existing file: jui-widgets.md

Chapter 17

Helpers

17.1 Helpers

> Note: This section is under development.

Helper classes typically contain static methods only and are used as follows:

```
use \yii\helpers\Html;
echo Html::encode('Test > test');
```

There are several classes provided by framework:

- ArrayHelper
- Console
- FileHelper
- Html
- HtmlPurifier
- Image
- Inflector
- Json
- Markdown
- Security
- StringHelper
- Url
- VarDumper

Error: not existing file: helper-array.md

Error: not existing file: helper-html.md

Error: not existing file: helper-url.md

Error: not existing file: helper-security.md