About the Tutorial

The Yii framework is an open-source PHP framework for rapidly-developing, modern Web applications. It is built around the Model-View-Controller composite pattern. Yii provides secure and professional features to create robust projects rapidly.

Audience

The Yii framework has a component-based architecture and a full solid caching support. Therefore, it is suitable for building all kinds of Web applications: forums, portals, content managements systems, RESTful services, e-commerce websites, and so forth.

Prerequisites

Yii is a pure OOP (Object-Oriented Programming) framework. Hence, it requires a basic knowledge of OOP. The Yii framework also uses the latest features of PHP, like traits and namespaces. The major requirements for Yii2 are PHP 5.4+ and a web server.

Copyright & Disclaimer

© Copyright 2016 by Tutorials Point (I) Pvt. Ltd.

All the content and graphics published in this e-book are the property of Tutorials Point (I) Pvt. Ltd. The user of this e-book is prohibited to reuse, retain, copy, distribute or republish any contents or a part of contents of this e-book in any manner without written consent of the publisher.

We strive to update the contents of our website and tutorials as timely and as precisely as possible, however, the contents may contain inaccuracies or errors. Tutorials Point (I) Pvt. Ltd. provides no guarantee regarding the accuracy, timeliness or completeness of our website or its contents including this tutorial. If you discover any errors on our website or in this tutorial, please notify us at contact@tutorialspoint.com
# Table of Contents

About the Tutorial ........................................................................................................... i
Audience........................................................................................................................... i
Prerequisites...................................................................................................................... i
Copyright & Disclaimer .................................................................................................. i
Table of Contents ........................................................................................................... ii

1. Yii – Overview ............................................................................................................. 1
   Core Features ............................................................................................................... 1
   Environment .................................................................................................................. 1

2. Yii – Installation ......................................................................................................... 4

3. Yii – Create Page ....................................................................................................... 6

4. Yii – Application Structure ....................................................................................... 8
   Yii2 – Objects .............................................................................................................. 9

5. Yii – Entry Scripts .................................................................................................... 11

6. Yii – Controllers ..................................................................................................... 14
   Understanding Actions .............................................................................................. 14
   Understanding Routes ............................................................................................... 18

7. Yii – Using Controllers ............................................................................................ 20

8. Yii – Using Actions .................................................................................................. 24
   Create a Standalone Action Class ............................................................................. 25
   Important Points ........................................................................................................ 29

9. Yii – Models ............................................................................................................... 30
   Attributes .................................................................................................................... 30
   Attribute Labels ......................................................................................................... 32
   Scenarios ...................................................................................................................... 34
   Massive Assignment .................................................................................................. 37
   Data Export ................................................................................................................ 38
   Important Points ........................................................................................................ 39

10. Yii – Widgets ............................................................................................................ 40
    Using Widgets ............................................................................................................ 40
    Creating Widgets .................................................................................................... 41
    Important Points ...................................................................................................... 44

11. Yii – Modules .......................................................................................................... 45
    Important Points ....................................................................................................... 48

12. Yii – Views ............................................................................................................... 49
    Creating Views ........................................................................................................ 49
    Accessing Data in Views ......................................................................................... 54

13. Yii – Layouts .......................................................................................................... 56
50. **Yii – Database Migration** .................................................................................................................. 212
    Creating a Migration ......................................................................................................................... 212

51. **Yii – Theming** .................................................................................................................................. 219

52. **Yii - RESTful APIs** ........................................................................................................................... 225
    Preparing the DB .............................................................................................................................. 225
    Installing Postman ............................................................................................................................ 227

53. **Yii – RESTful APIs in Action** .......................................................................................................... 228

54. **Yii – Fields** ...................................................................................................................................... 234
    Customizing Actions .......................................................................................................................... 237
    Handling Errors ................................................................................................................................. 238

55. **Yii – Testing** .................................................................................................................................... 239
    Preparing for the Tests ....................................................................................................................... 239
    Fixtures ............................................................................................................................................. 240
    Unit Tests .......................................................................................................................................... 240
    Functional Tests ............................................................................................................................... 242

56. **Yii – Caching** ................................................................................................................................... 244
    Preparing the DB .............................................................................................................................. 244
    Data Caching .................................................................................................................................... 246
    Query Caching .................................................................................................................................. 250

57. **Fragment Caching** ........................................................................................................................ 252
    Page Caching .................................................................................................................................... 253
    HTTP Caching ................................................................................................................................... 255

58. **Yii – Aliases** ..................................................................................................................................... 257

59. **Yii – Logging** .................................................................................................................................... 259

60. **Yii – Error Handling** ....................................................................................................................... 265

61. **Yii – Authentication** ........................................................................................................................ 271

62. **Yii – Authorization** .......................................................................................................................... 278
    Passwords .......................................................................................................................................... 279

63. **Yii – Localization** ............................................................................................................................ 281

64. **Yii – Gii** .......................................................................................................................................... 286
    Preparing the DB .............................................................................................................................. 287

65. **Gii – Creating a Model** .................................................................................................................... 289
    Generating CRUD .............................................................................................................................. 290

66. **Gii – Generating Controller** ............................................................................................................ 292
    Form Generation ............................................................................................................................... 293

---

Save Data to Database ..................................................................................................................................... 210
67. Gii – Generating Module
The **Yii** framework is an open-source PHP framework for rapidly-developing, modern Web applications. It is built around the Model-View-Controller composite pattern.

Yii provides secure and professional features to create robust projects rapidly. The Yii framework has a component-based architecture and a full solid caching support. Therefore, it is suitable for building all kinds of Web applications: forums, portals, content managements systems, RESTful services, e-commerce websites, and so forth. It also has a code generation tool called Gii that includes the full CRUD(Create-Read-Update-Delete) interface maker.

### Core Features

The core features of Yii are as follows:

- Yii implements the MVC architectural pattern.
- It provides features for both relational and NoSQL databases.
- Yii never over-designs things for the sole purpose of following some design pattern.
- It is extremely extensible.
- Yii provides multi-tier caching support.
- Yii provides RESTful API development support.
- It has high performance.

Overall, if all you need is a neat interface for the underlying database, then Yii is the right choice. Currently, Yii has two versions: 1.1 and 2.0.

Version 1.1 is now in maintenance mode and Version 2 adopts the latest technologies, including Composer utility for package distribution, PSR levels 1, 2, and 4, and many PHP 5.4+ features. It is version 2 that will receive the main development effort over the next few years.

Yii is a pure OOP (Object-Oriented Programming) framework. Hence, it requires a basic knowledge of OOP. The Yii framework also uses the latest features of PHP, like traits and namespaces. It would be easier for you to pick up Yii 2.0 if you understand these concepts.

### Environment

The major requirements for Yii2 are **PHP 5.4+** and a **web server**. Yii is a powerful console tool, which manages database migrations, asset compilation, and other stuff. It is recommended to have a command line access to the machine where you develop your application.
For development purpose, we will use:

- Linux Mint 17.1
- PHP 5.5.9
- PHP built-in web server

**Pre-installation check**

To check whether your local machine is good to go with the latest Yii2 version, do the following:

1. Install the latest php version:
   
   ```bash
   sudo apt-get install php5
   ```

2. Install the latest mysql version:
   
   ```bash
   sudo apt-get install mysql-server
   ```

3. Download the Yii2 basic application template:
   
   ```bash
   composer create-project --prefer-dist --stability=dev yiisoft/yii2-app-basic basic
   ```

4. To start a PHP built-in server, inside the `basic` folder run:
   
   ```bash
   php -S localhost:8080
   ```

There is a useful script, `requirements.php`. It checks whether your server meets the requirements to run the application. You can find this script in the root folder of your application.
If you type http://localhost:8080/requirements.php in the address bar of the web browser, the page looks like as shown in the following screenshot:
2. Yii – Installation

The most straightforward way to get started with Yii2 is to use the basic application template provided by the Yii2 team. This template is also available through the Composer tool.

1. Find a suitable directory in your hard drive and download the Composer PHAR (PHP archive) via the following command:

```
curl -sS https://getcomposer.org/installer | php
```

2. Then move this archive to the bin directory.

```
mv composer.phar /usr/local/bin/composer
```

3. With the Composer installed, you can install Yii2 basic application template. Run these commands:

```
composer global require "fxp/composer-asset-plugin:^1.1.1"
composer create-project --prefer-dist yiisoft/yii2-app-basic helloworld
```

The first command installs the composer asset plugin, which manages npm and bower dependencies. The second command installs Yii2 basic application template in a directory called **helloworld**.

4. Now open the **helloworld** directory and launch the web server built into PHP.

```
php -S localhost:8080 -t web
```
5. Then open **http://localhost:8080** in your browser. You can see the welcome page:
Now we are going to create a “Hello world” page in your application. To create a page, we must create an action and a view.

Actions are declared in controllers. The end user will receive the execution result of an action.

1. Declare the speak action in the existing SiteController, which is defined in the class file controllers/SiteController.php.

```php
<?php
namespace app\controllers;
use Yii;
use yii\filters\AccessControl;
use yii\web\Controller;
use yii\filters\VerbFilter;
use app\models\LoginForm;
use app\models\ContactForm;

class SiteController extends Controller
{
    /* other code */
    public function actionSpeak($message = "default message")
    {
        return $this->render("speak", ['message' => $message]);
    }
}
?>
```

We defined the speak action as a method called actionSpeak. In Yii, all action methods are prefixed with the word action. This is how the framework differentiates action methods from non-action ones. If an action ID requires multiple words, then they will be concatenated by dashes. Hence, the action ID add-post corresponds to the action method actionAddPost.

In the code given above, the `out` function takes a GET parameter, $message. We also call a method named `render` to render a view file called speak. We pass the message parameter to the view. The rendering result is a complete HTML page.
**View** is a script that generates a response's content. For the speak action, we create a speak view that prints our message. When the render method is called, it looks for a PHP file names as `view/controllerID/viewName.php`.

2. Therefore, inside the views/site folder create a file called `speak.php` with the following code:

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

    ?></p>
<?php echo Html::encode($message); ?>
```

Note that we HTML-encode the message parameter before printing to avoid XSS attack.

3. Type the following in your web browser

You will see the following window:

The 'r' parameter in the URL stands for route. The route's default format is `controllerID/actionID`. In our case, the route site/speak will be resolved by the `SiteController` class and the speak action.
4. **Yii – Application Structure**

There is only one folder in the overall code base that is publicly available for the web server. It is the web directory. Other folders outside the web root directory are out of reach for the web server.

**Note:** All project dependencies are located in the `composer.json` file. Yii2 has a few important packages that are already included in your project by Composer. These packages are the following:

- Gii – The code generator tool
- The debug console
- The Codeception testing framework
- The SwiftMailer library
- The Twitter Bootstrap UI library

The first three packages are only useful in the development environment.

Yii2’s application structure is precise and clear. It contains the following folders:

- **Assets:** This folder includes all .js and .css files referenced in the web page.
- **Commands:** This folder includes the controllers that can be used from the terminal.
- **Config:** This folder contains `config` files for managing database, application and application parameters.
- **Mail:** This folder includes the mail layout.
- **Models:** This folder includes the models used in the application.
- **Runtime:** This folder is for storing runtime data.
- **Tests:** This folder includes all the tests (acceptance, unit, functional).
- **Vendor:** This folder contains all the third-party packages managed by Composer.
- **Views:** This folder is for views, that are displayed by the controllers. The `layout` folder is a for page template.
- **Web:** The entry point from web.
Application Structure

Following is the diagrammatic representation of the application structure.

Yii2 – Objects

The following list contains all Yii2's objects:

Models, Views, and Controllers

Models are for data representation (usually from the database). View are for displaying the data. Controllers are for processing requests and generating responses.
Components
To create a reusable functionality, the user can write his own components. Components are just objects that contain logic. For example, a component could be a weight converter.

Application components
These are objects that instanced just one time in the whole application. The main difference between Components and Application components is that the latter can have only one instance in the whole application.

Widgets
Widgets are reusable objects containing both logic and rendering code. A widget could be, for example, a gallery slider.

Filters
Filters are objects that run before or after the execution of the Controller actions.

Modules
You can consider Modules as reusable subapps, containing Models, Views, Controllers, and so forth.

Extensions
Extensions are packages that can be managed by the Composer.
5. Yii – Entry Scripts

Entry scripts are responsible for starting a request handling cycle. They are just PHP scripts accessible by users.

The following illustration shows the structure of an application:

Web application (as well as console application) has a single entry script. The End user makes request to the entry script. Then the entry script instantiates application instances and forwards requests to them.

Entry script for a console application is usually stored in a project base path and named as yii.php. Entry script for a web application must be stored under a web accessible directory. It is often called index.php.
The Entry scripts do the following:

- Define constants.
- Register Composer autoloader.
- Include Yii files.
- Load configuration.
- Create and configure an application instance.
- Process the incoming request.

The following is the entry script for the **basic application** template:

```php
<?php
//defining global constants
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 files
require(__DIR__ . '/../vendor/yiisoft/yii2/Yii.php');

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

//create, config, and process request
(new yii\web\Application($config))->run();
?>
```

The following is the entry script for the **console** application:

```
#!/usr/bin/env php
<?php
/**
 * Yii console bootstrap file.
 * @link http://www.yiiframework.com/
 */
```

```
The best place for defining global constants is entry scripts. There are three supported by Yii constants:

- **YII_DEBUG**: Defines whether you are in debug mode or not. If set to true, then we will see more log data and detail error call stack.

- **YII_ENV**: Defines the environment mode. The default value is prod. Available values are prod, dev, and test. They are used in configuration files to define, for example, a different DB connection (local and remote) or other values.

- **YII_ENABLE_ERROR_HANDLER**: Specifies whether to enable the default Yii error handler.

To define a global constant the following code is used:

```php
//defining global constants
defined('YII_DEBUG') or define('YII_DEBUG', true);
```
which is equivalent to:

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

**Note:** The global constants should be defined at the beginning of an entry script in order to take effect when other PHP files are included.
Controllers are responsible for processing requests and generating responses. After user’s request, the controller will analyze request data, pass them to model, then insert the model result into a view, and generate a response.

**Understanding Actions**

Controllers include actions. They are the basic units that user can request for execution. A controller can have one or several actions.

Let us have a look at the `SiteController` of the basic application template:

```php
<?php
namespace app\controllers;
use yii;
use yii\filters\AccessControl;
use yii\web\Controller;
use yii\filters\VerbFilter;
use app\models\LoginForm;
use app\models\ContactForm;
class SiteController extends Controller
{
    public function behaviors()
    {
        return [
            'access' => [
                'class' => AccessControl::className(),
                'only' => ['logout'],
                'rules' => [
                    'actions' => ['logout'],
                    'allow' => true,
                    'roles' => ['@'],
                ],
            ],
        ];
    }
}
```
'verbs' => [
    'class' => VerbFilter::className(),
    'actions' => [
        'logout' => ['post'],
    ],
],

public function actions()
{
    return [
        'error' => [
            'class' => 'yii\web\ErrorAction',
        ],
        'captcha' => [
            'class' => 'yii\captcha\CaptchaAction',
            'fixedVerifyCode' => YII_ENV_TEST ? 'testme' : null,
        ],
    ];
}

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

public function actionLogin()
{
    if (!Yii::$app->user->isGuest) {
        return $this->goHome();
    }
}
Yii

```php
$model = new LoginForm();
if ($model->load(Yii::$app->request->post()) && $model->login()) {
    return $this->goBack();
}
return $this->render('login', [
    'model' => $model,
]);
}
public function actionLogout()
{
    Yii::$app->user->logout();
    return $this->goHome();
}
public function actionContact()
{
    //load ContactForm model
    $model = new ContactForm();
    //if there was a POST request, then try to load POST data into a model
    if ($model->load(Yii::$app->request->post()) && $model->contact(Yii::$app->params['adminEmail'])) {
        Yii::$app->session->setFlash('contactFormSubmitted');
        return $this->refresh();
    }
    return $this->render('contact', [
        'model' => $model,
    ]);}
public function actionAbout()
{
    return $this->render('about');
}
```
public function actionSpeak($message = "default message")
{
    return $this->render("speak", ['message' => $message]);
}
?>

Run the basic application template using PHP built-in server and go to the web browser at http://localhost:8080/index.php?r=site/contact. You will see the following page:

When you open this page, the contact action of the SiteController is executed. The code first loads the ContactForm model. Then it renders the contact view and passes the model into it.
If you fill in the form and click the submit button, you will see the following:

Notice that this time the following code is executed:
If there was a POST request, we assign the POST data to the model and try to send an email. If we success then we set a flash message with the text “Thank you for contacting us. We will respond to you as soon as possible.” and refresh the page.

**Understanding Routes**

In the above example, in the URL, `http://localhost:8080/index.php?r=site/contact`, the route is `site/contact`. The contact action (`actionContact`) in the `SiteController` will be executed.

A route consists of the following parts:

- **moduleId**: If the controller belongs to a module, then this part of the route exists.
- **controllerId** (`site` in the above example): A unique string that identifies the controller among all controllers within the same module or application.
- **actionId** (`contact` in the above example): A unique string that identifies the action among all actions within the same controller.

The format of the route is `controllerId/actionId`. If the controller belongs to a module, then it has the following format: `moduleId/controllerId/actionId`. 
Controllers in web applications should extend from `yii\web\Controller` or its child classes. In console applications, they should extend from `yii\console\Controller` or its child classes.

Let us create an example controller in the `controllers` folder.

1. Inside the `Controllers` folder, create a file called `ExampleController.php` with the following code:

```php
<?php
namespace app\controllers;
use yii\web\Controller;
class ExampleController extends Controller
{
    public function actionIndex()
    {
        $message = "index action of the ExampleController";
        return $this->render("example",
            ['message' => $message
        ]);  
    }
}
?>
```

2. Create an example view in the `views/example` folder. Inside that folder, create a file called `example.php` with the following code:

```php
<?php
echo $message;
?>
```

Each application has a default controller. For web applications, the site is the controller, while for console applications it is `help`. Therefore, when the `http://localhost:8080/index.php` URL is opened, the site controller will handle the request. You can change the default controller in the application configuration.
Consider the given code:

```
'defaultRoute' => 'main'
```

3. Add the above code to the following `config/web.php`:

```php
<?php
$params = require(__DIR__ . '/params.php');
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
    'components' => [
        'request' => [
            'cookieValidationKey' => 'ymoaYrebZHa8gURuo6ioHG1K9fLXCKjo',
        ],
        'cache' => [
            'class' => 'yii\caching\FileCache',
        ],
        'user' => [
            'identityClass' => 'app\models\User',
            'enableAutoLogin' => true,
        ],
        'errorHandler' => [
            'errorAction' => 'site/error',
        ],
        'mailer' => [
            'class' => 'yii\swiftmailer\Mailer',
            'useFileTransport' => true,
        ],
    ],
];
```
'log' => [  
    'traceLevel' => YII_DEBUG ? 3 : 0,
    'targets' => [
        [
            'class' => 'yii\log\FileTarget',
            'levels' => ['error', 'warning'],
        ],
    ],
    'db' => require(__DIR__ . '/db.php'),
],
// changing the default controller
'defaultRoute' => 'example',
'params' => $params,
];
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
    $config['bootstrap'][] = 'debug';
    $config['modules']['debug'] = [
        'class' => 'yii\debug\Module',
    ];
    $config['bootstrap'][] = 'gii';
    $config['modules']['gii'] = [
        'class' => 'yii\gii\Module',
    ];
}
return $config;
?>

4. Type http://localhost:8080/index.php in the address bar of the web browser, you will see that the default controller is the example controller.
Note: The Controller IDs should contain English letters in lower case, digits, forward slashes, hyphens, and underscores.

To convert the controller ID to the controller class name, you should do the following:

- Take the first letter from all words separated by hyphens and turn it into uppercase.
- Remove hyphens.
- Replace forward slashes with backward ones.
- Add the Controller suffix.
- Prepend the controller namespace.

Examples

- page becomes `app\controllers\PageController`
- post-article becomes `app\controllers\PostArticleController`
- user/post-article becomes `app\controllers\user\PostArticleController`
- userBlogs/post-article becomes `app\controllers\userBlogs\PostArticleController`
To create an action in a controller class, you should define a public method whose name starts with the word `action`. The return data of an action represents the response to be sent to the end user.

1. Let us define the hello-world action in our `ExampleController`.

```php
<?php
namespace app\controllers;
use yii\web\Controller;
class ExampleController extends Controller
{
    public function actionIndex(){
        $message = "index action of the ExampleController";
        return $this->render("example", [
            'message' => $message
        ]);}
    }
    public function actionHelloWorld(){
        return "Hello world!";
    }
}
?>
```
2. Type `http://localhost:8080/index.php?r=example/hello-world` in the address bar of the web browser. You will see the following:

![Hello world!](image)

Action IDs are usually verbs, such as create, update, delete and so on. This is because actions are often designed to perform a particular change if a resource.

Action IDs should contain only these characters: English letters in lower case, digits, hyphens, and underscores.

There are two types of actions: inline and standalone.

Inline actions are defined in the controller class. The names of the actions are derived from action IDs this way:

- Turn the first letter in all words of the action ID into uppercase.
- Remove hyphens.
- Add the action prefix.

**Examples:**

- index becomes actionIndex
- hello-world (as in the example above) becomes actionHelloWorld

If you plan to reuse the same action in different places, you should define it as a standalone action.

**Create a Standalone Action Class**

To create a standalone action class, you should extend yiiase\Action or a child class, and implement a `run()` method.
1. Create a components folder inside your project root. Inside that folder create a file called `GreetingAction.php` with the following code:

```php
<?php
namespace app\components;
use yii\base\Action;
class GreetingAction extends Action
{
    public function run()
    {
        return "Greeting";
    }
}
?>
```

We have just created a reusable action. To use it in our `ExampleController`, we should declare our action in the action map by overriding the `actions()` method.

2. Modify the `ExampleController.php` file this way:

```php
<?php
namespace app\controllers;
use yii\web\Controller;
class ExampleController extends Controller
{
    public function actions()
    {
        return [
            'greeting' => 'app\components\GreetingAction',
        ];
    }
    public function actionIndex()
    {
        $message = "index action of the ExampleController";
        return $this->render("example",[
            'message' => $message
        ]);
The `actions()` method returns an array whose values are class names and keys are action IDs.

3. Go to `http://localhost:8080/index.php?r=example/greeting`. You will see the following output:

![Image](http://localhost:8080/index.php?r=example/greeting)

**Greeting**

4. You can also use actions to redirect users to other URLs. Add the following action to the `ExampleController.php`:

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


The action methods can take parameters, called *action parameters*. Their values are retrieved from `$_GET` using the parameter name as the key.

5. Add the following action to our example controller:
public function actionTestParams($first, $second)
{
    return "$first $second";
}

6. Type the URL http://localhost:8080/index.php?r=example/test-params&first=hello&second=world in the address bar of your web browser, you will see the following output:

   hello world

Each controller has a default action. When a route contains the controller ID only, it means that the default action is requested. By default, the action is index. You can easily override this property in the controller.

7. Modify our ExampleController this way:

<?php
namespace app\controllers;
use yii\web\Controller;
class ExampleController extends Controller{
    public $defaultAction = "hello-world";
    /* other actions */
}
?>

8. Now, if you go to http://localhost:8080/index.php?r=example, you will see the following:
To fulfill the request, the controller will undergo the following lifecycle:

- The `yii\base\Controller::init()` method is called.
- The controller creates an action based on the action ID.
- The controller sequentially calls the `beforeAction()` method of the web application, module, and the controller.
- The controller runs the action.
- The controller sequentially calls the `afterAction()` method of the web application, module, and the controller.
- The application assigns action result to the response.

**Important Points**

The Controllers should:

- Be very thin. Each action should contain only a few lines of code.
- Use Views for responses.
- Not embed HTML.
- Access the request data.
- Call methods of models.
- Not process the request data. These should be processed in the model.
Models are objects representing business logic and rules. To create a model, you should extend the \yii\base\Model class or its subclasses.

### Attributes

Attributes represent the business data. They can be accessed like array elements or object properties. Each attribute is a publicly accessible property of a model. To specify what attributes a model possesses, you should override the `yii\base\Model::attributes()` method.

Let us have a look at the **ContactForm** model of the basic application template.

```php
<?php
namespace app\models;
use yii;
use yii\base\Model;
/**
 * ContactForm is the model behind the contact form.
 */
class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;
    public $verifyCode;
    /**
     * @return array the validation rules.
     */
    public function rules()
    {
        return [
```
// name, email, subject and body are required
[['name', 'email', 'subject', 'body'], 'required'],
// email has to be a valid email address
['email', 'email'],
// verifyCode needs to be entered correctly
['verifyCode', 'captcha'],
];
}
/**
 * @return array customized attribute labels
 */
public function attributeLabels()
{
    return [
        'verifyCode' => 'Verification Code',
    ];
}
/**
 * Sends an email to the specified email address using the information collected
 * by this model.
 * @param  string  $email the target email address
 * @return boolean whether the model passes validation
 */
public function contact($email)
{
    if ($this->validate()) {
        Yii::$app->mailer->compose()
            ->setTo($email)
            ->setFrom([$this->email => $this->name])
            ->setSubject($this->subject)
            ->setTextBody($this->body)
            ->send();

        return true;
    }
1. Create a function called `actionShowContactModel` in the `SiteController` with the following code:

```php
public function actionShowContactModel(){
    $mContactForm = new \app\models\ContactForm();
    $mContactForm->name = "contactForm";
    $mContactForm->email = "user@gmail.com";
    $mContactForm->subject = "subject";
    $mContactForm->body = "body";
    var_dump($mContactForm);
}
```

In the above code, we define the `ContactForm` model, set attributes, and display the model on the screen.

2. Now, if you type `http://localhost:8080/index.php?r=site/show-contact-model` in the address bar of the web browser, you will see the following:

```php
object(app\models\ContactForm)[39]
  public 'name' => string 'contactForm' (length=11)
  public 'email' => string 'user@gmail.com' (length=14)
  public 'subject' => string 'subject' (length=7)
  public 'body' => string 'body' (length=4)
  public 'verifyCode' => null
  private '_errors' (yii\base\Model) => null
  private '_validators' (yii\base\Model) => null
  private '_scenario' (yii\base\Model) => string 'default' (length=7)
  private '_events' (yii\base\Component) =>
    array (size=0)
      empty
  private '_behaviors' (yii\base\Component) => null
```
If your model extends from `yii\base\Model`, then all its member variables (public and non-static) are attributes. There are five attributes in the `ContactForm` model: name, email, subject, body, `verifyCode` and you can easily add new ones.

**Attribute Labels**

You often need to display labels associated with attributes. By default, attribute labels are automatically generated by the `yii\base\Model::generateAttributeLabel()` method. To manually declare attribute labels, you may override the `yii\base\Model::attributeLabels()` method.

1. If you open `http://localhost:8080/index.php?r=site/contact`, you will see the following page:

![Contact Form Image](image)

Note that the attribute labels are the same as their names.

2. Now, modify the `attributeLabels` function in the `ContactForm` model in the following way:

```php
public function attributeLabels()
{
    return [
        'name' => 'name overridden',
    ];
}
```
'email' => 'email overridden',
'subject' => 'subject overridden',
'body' => 'body overridden',
'verlifyCode' => 'verifyCode overridden',
];}
}

3. If you open **http://localhost:8080/index.php?r=site/contact** again, you will notice that the labels have changed as shown in the following image.

Scenarios

You can use a model in different scenarios. For example, when a guest wants to send a contact form, we need all the model attributes. When a user wants to do the same thing, he is already logged in, so we do not need his name, as we can easily take it from the DB.

To declare scenarios, we should override the **scenarios()** function. It returns an array whose keys are the scenario names and values are active attributes. Active attributes are the ones to validate. They can also be massively assigned.

1. Modify the **ContactForm** model in the following way:
<?php
namespace app\models;
use Yii;
use yii\base\Model;
/**
 * ContactForm is the model behind the contact form.
 */
class ContactForm extends Model
{
    public $name;
    public $email;
    public $subject;
    public $body;
    public $verifyCode;
    const SCENARIO_EMAIL_FROM_GUEST = 'EMAIL_FROM_GUEST';
    const SCENARIO_EMAIL_FROM_USER = 'EMAIL_FROM_USER';
    public function scenarios()
    {
        return [
            self::SCENARIO_EMAIL_FROM_GUEST => ['name', 'email', 'subject', 'body', 'verifyCode'],
            self::SCENARIO_EMAIL_FROM_USER => ['email', 'subject', 'body', 'verifyCode'],
        ];
    }
    /**
     * @return array the validation rules.
     */
    public function rules()
    {
        return [
            // name, email, subject and body are required
            [['name', 'email', 'subject', 'body'], 'required'],
        ];
    }
}
// email has to be a valid email address
['email', 'email'],
// verifyCode needs to be entered correctly
['verifyCode', 'captcha'],
];
}

/**
 * @return array customized attribute labels
 */
public function attributeLabels()
{
    return [
        'name' => 'name overridden',
        'email' => 'email overridden',
        'subject' => 'subject overridden',
        'body' => 'body overridden',
        'verifyCode' => 'verifyCode overridden',
    ];
}

/**
 * Sends an email to the specified email address using the information collected
 * by this model.
 * @param  string  $email the target email address
 * @return boolean whether the model passes validation
 */
public function contact($email)
{
    if ($this->validate()) {
        Yii::$app->mailer->compose()
            ->setTo($email)
            ->setFrom([$this->email => $this->name])
            ->setSubject($this->subject)
We have added two scenarios. One for the guest and another for authenticated user. When the user is authenticated, we do not need his name.

2. Now, modify the `actionContact` function of the `SiteController`:

```php
public function actionContact()
{
    $model = new ContactForm();
    $model->scenario = ContactForm::SCENARIO_EMAIL_FROM_GUEST;
    if ($model->load(Yii::$app->request->post()) && $model->contact(Yii::$app->params['adminEmail'])) {
        Yii::$app->session->setFlash('contactFormSubmitted');

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

3. Type `http://localhost:8080/index.php?r=site/contact` in the web browser. You will notice that currently, all model attributes are required.
4. If you change the scenario of the model in the `actionContact`, as given in the following code, you will find that the name attribute is no longer required.

```php
$model->scenario = ContactForm::SCENARIO_EMAIL_FROM_USER;
```
Massive Assignment

Massive assignment is a convenient way of creating a model from multiple input attributes via a single line of code.

The lines of code are:

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

The above given lines of code are equivalent to:

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

The former is much cleaner. Notice that massive assignment only applies to the safe attributes. They are just the current scenario attributes listed in the `scenario()` function.

Data Export

The Models often need to be exported in different formats. To convert the model into an array, modify the `actionShowContactModel` function of the `SiteController`:

```php
public function actionShowContactModel(){
    $mContactForm = new \app\models\ContactForm();
    $mContactForm->name = "contactForm";
    $mContactForm->email = "user@gmail.com";
    $mContactForm->subject = "subject";
    $mContactForm->body = "body";
    var_dump($mContactForm->attributes);
}
```
Type **http://localhost:8080/index.php?r=site/show-contact-model** in the address bar and you will see the following:

![Browser output](image)

```
array (size=5)
  'name' => string 'contactForm' (length=11)
  'email' => string 'user@gmail.com' (length=14)
  'subject' => string 'subject' (length=7)
  'body' => string 'body' (length=4)
  'verifyCode' => null
```

To convert the Model to the **JSON** format, modify the **actionShowContactModel** function in the following way:

```
public function actionShowContactModel(){
    $mContactForm = new \app\models\ContactForm();
    $mContactForm->name = "contactForm";
    $mContactForm->email = "user@gmail.com";
    $mContactForm->subject = "subject";
    $mContactForm->body = "body";
    return \yii\helpers\Json::encode($mContactForm);
}
```

**Browser output:**

```
{"name":"contactForm","email":"user@gmail.com","subject":"subject","body":"body","verifyCode":null}
```
Important Points

Models are usually much faster than controllers in a well-designed application. Models should:

- Contain business logic.
- Contain validation rules.
- Contain attributes.
- Not embed HTML.
- Not directly access requests.
- Not have too many scenarios.
A widget is a reusable client-side code, which contains HTML, CSS, and JS. This code includes minimal logic and is wrapped in a `yiiase\Widget` object. We can easily insert and apply this object in any view.

1. To see widgets in action, create an `actionTestWidget` function in the `SiteController` with the following code:

   ```php
   public function actionTestWidget(){
       return $this->render('testwidget');
   }
   ```

   In the above example, we just returned a View called “testwidget”.

2. Now, inside the views/site folder, create a View file called `testwidget.php`

   ```php
   use yii\bootstrap\Progress;
   
   <?= Progress::widget(['percent' => 60, 'label' => 'Progress 60%']) ?>
   ```

3. If you go to `http://localhost:8080/index.php?r=site/test-widget`, you will see the progress bar widget:

   ![Progress bar widget](image)

**Using Widgets**

To use a widget in a View, you should call the `yiiase\Widget::widget()` function. This function takes a configuration array for initializing the widget. In the previous example, we inserted a progress bar with percent and labelled parameters of the configuration object.
Some widgets take a block of content. It should be enclosed between `yiiase\Widget::begin()` and `yiiase\Widget::end()` functions. For example, the following widget displays a contact form:

```php
<?php $form = ActiveForm::begin(['id' => 'contact-form']); ?>
    <?= $form->field($model, 'name') ?>
    <?= $form->field($model, 'email') ?>
    <?= $form->field($model, 'subject') ?>
    <?= $form->field($model, 'body')->textarea(['rows' => 6]) ?>
    <?= $form->field($model, 'verifyCode')->widget(Captcha::className(), ['template' => '<div class="row"><div class="col-lg-3">{image}</div><div class="col-lg-6">{input}</div></div>'],)]?>
    <div class="form-group">
        <?= Html::submitButton('Submit', ['class' => 'btn btn-primary', 'name' => 'contact-button']) ?>
    </div>
<?php ActiveForm::end(); ?>
```

**Creating Widgets**

To create a widget, you should extend from `yiiase\Widget`. Then you should override the `yiiase\Widget::init()` and `yiiase\Widget::run()` functions. The `run()` function should return the rendering result. The `init()` function should normalize the widget properties.

1. Create a components folder in the project root. Inside that folder, create a file called `FirstWidget.php` with the following code:

```php
<?php
namespace app\components;
use yii\base\Widget;

class FirstWidget extends Widget
{
    public $mes;
    public function init()
    {
```
Yii

```php
parent::init();
if ($this->mes === null) {
    $this->mes = 'First Widget';
}

public function run()
{
    return '<h1>$this->mes</h1>";
}
?>
```

2. **Modify** the `testwidget` view in the following way:

```php
<?php
use app\components\FirstWidget;
?>
<?= FirstWidget::widget() ?>
```  


![My Company](image)

First Widget
4. To enclose the content between the `begin()` and `end()` calls, you should modify the `FirstWidget.php` file:

```php
<?php
namespace app\components;
use yii\base\Widget;
class FirstWidget extends Widget
{
    public function init()
    {
        parent::init();
        ob_start();
    }
    public function run()
    {
        $content = ob_get_clean();
        return "<h1>$content</h1>";
    }
}
?>
```

5. Now h1 tags will surround all the content. Notice that we use the `ob_start()` function to buffer the output. Modify the `testwidget` view as given in the following code.

```php
<?php
use app\components\FirstWidget;
?>
<?php FirstWidget::begin(); ?>
    First Widget in H1
<?php FirstWidget::end(); ?>
```

You will see the following output:
Important Points

Widgets should:

- Be created following the MVC pattern. You should keep presentation layers in views and logic in widget classes.

- Be designed to be self-contained. The end developer should be able to design it into a View.
A module is an entity that has its own models, views, controllers, and possibly other modules. It is practically an application inside the application.

1. Create a folder called **modules** inside your project root. Inside the modules folder, create a folder named **hello**. This will be the basic folder for our Hello module.

2. Inside the **hello** folder, create a file **Hello.php** with the following code:

```php
<?php
namespace app\modules\hello;

class Hello extends yii\base\Module
{
    public function init()
    {
        parent::init();
    }
}
?>
```

We have just created a module class. This should be located under the module's base path. Every time a module is accessed, an instance of the correspondent module class is created. The **init()** function is for initializing the module's properties.

3. Now, add two more directories inside the **hello** folder: controllers and views. Add a **CustomController.php** file to the controller's folder:

```php
<?php
namespace app\modules\hello\controllers;

use yii\web\Controller;

class CustomController extends Controller
{
    public function actionGreet()
    {
        return $this->render('greet');
    }
}
```
When creating a module, a convention is to put the controller classes into the controller’s directory of the module's base path. We have just defined the `actionGreet` function, that just returns a `greet` view.

Views in the module should be put in the views folder of the module's base path. If views are rendered by a controller, they should be located in the folder corresponding to the `controllerID`. Add `custom` folder to the `views` folder.

4. Inside the custom directory, create a file called `greet.php` with the following code:

```php
<h1>Hello world from custom module!</h1>
```

We have just created a View for our `actionGreet`. To use this newly created module, we should configure the application. We should add our module to the modules property of the application.

5. Modify the `config/web.php` file:

```php
<?php
$params = require(__DIR__ . '/params.php);
$config = [
    'id' => 'basic',
    'basePath' => dirname(__DIR__),
    'bootstrap' => ['log'],
    'components' => [
        'request' => [],
        // !!! insert a secret key in the following (if it is empty) - this is required by cookie validation
        'cookieValidationKey' => 'ymoaYrebZHa8gURuolioHG1K8fLXCkj0',
    ],
    'cache' => [
        'class' => 'yii\caching\FileCache',
    ],
    'user' => [
        'identityClass' => 'app\models\User',
    ],
];
```
'enableAutoLogin' => true,
],
'errorHandler' => [
'errorAction' => 'site/error',
],
'mailer' => [
'class' => 'yii\swiftmailer\Mailer',
// send all mails to a file by default. You have to set
// 'useFileTransport' to false and configure a transport
// for the mailer to send real emails.
'useFileTransport' => true,
],
'log' => [
'traceLevel' => YII_DEBUG ? 3 : 0,
'targets' => [
[
'class' => 'yii\log\FileTarget',
'levels' => ['error', 'warning'],
],
],
],
'db' => require(__DIR__ . '/db.php'),
],
'modules' => [
'hello' => [
'class' => 'app\modules\hello\Hello',
],
],
'params' => $params,
];
if (YII_ENV_DEV) {
    // configuration adjustments for 'dev' environment
$config['bootstrap'][] = 'debug';
$config['modules']['debug'] = [
    'class' => 'yii\debug\Module',
];
$config['bootstrap'][] = 'gii';
$config['modules']['gii'] = [
    'class' => 'yii\gii\Module',
];
}
return $config;
?>

A route for a module's controller must begin with the module ID followed by the controller ID and action ID.

6. To run the **actionGreet** in our application, we should use the following route:

```
hello/custom/greet
```

Where hello is a module ID, custom is a **controller ID** and greet is an **action ID**.

7. Now, type **http://localhost:8080/index.php?r=hello/custom/greet** and you will see the following output:

```
Hello world from custom module!
```

**Important Points**

Modules should:
- Be used in large applications. You should divide its features into several groups. Each feature group can be developed as a module.

- Be reusable. Some commonly used features, as SEO management or blog management, can be developed as modules, so that you can easily reuse them in future projects.
End of ebook preview
If you liked what you saw...
Buy it from our store @ https://store.tutorialspoint.com