**Advanced Programming – Assignment # 1**

**Name:** Mohammad Hamza Zakir **Section:** BSCS-7B **CMS ID:** 210632

I have been working on a PHP framework called Yii (Yes, it is!). Hailed as a fast, dynamic, and efficient framework, Yii essentially allows for rapid application development (RAD) and enables you to build applications that are fairly stable and scalable.

Like every other PHP framework, it is based on the MVC (Model-View-Controller) architecture, whereby a user requests data from the “controller” (controls the logic) via the “view” (user interface), and the “controller” then manipulates the data in the “model” accordingly before sending the final result back to the user’s “view”.

Even though Yii is accompanied with a considerably steep learning curve for a beginner, its easy installation process and immense community support make it easier to understand and use as you move along. It features high security and extensibility thereby making it ideal for projects like ecommerce and CMS. Moreover, as far as the database side of things is concerned, it helps developers by preventing the need to continuously write SQL statements, as it allows you to model the database data in terms of objects.

**Introduction**

To start with, I installed Yii on my computer via Composer, which is a package manager that also allows you to download further plugins/extensions associated with the framework via simple command line statement.

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

The above command is what I entered in the CMD terminal to install Yii via Composer. This provided me with a directory called **basic** that contains the application code and template. In order to access this template, all I had to do was to run the following command:

php yii serve

This configured the built-in PHP server, and when I entered the URL <http://localhost:8080/basic/web/index.php> into the browser, the basic app template popped up. It is simple web app that contains four pages: Home, About, Contact, and Login. All of these pages consist of a common header and footer, with the header comprising of the menu that allows you to navigate between the pages.

Initially, I merely explored this basic template itself. The About page simply states that it can be customized by modifying the about.php file in the site folder of the basic directory. The Contact page consists of an entry form and a submit button, while the Login page also consists of a form that requires username and password with a Login button below.

I started out by modifying the about.php file in order to change the title name from “About” to “Hamza Page”, and changing the present welcome statement from “This is the About page. You may modify the following file to customize its content:” to “My name is Hamza.”

**Code structure and Database**

The overall code structure echoes the MVC architecture, as there is a folder called *controllers* containing all the Controller classes, a folder called *models* containing all the Model classed, and a folder called *views* containing all the View classes.

Yii has support for various database applications, and if you want to fetch data from a database in an application, all you have to do is to configure the database connection, create an Active Record class to fetch and represent that data, create an action to manipulate the data, and finally a view to represent the end result to the users.

It is worth noting that installing the PDO (PHP Data Objects) extension is imperative to be able to deal with and configure relational databases; once your database is created, you head over to config/db.php and change the parameters like username, password and class so that they match your database’s credentials. This is how the database connection gets configured, and you can then proceed to fetch data and represent it.

**“Hello World” and working with forms**

After my initial exploration of the template, code structure, and database was complete, I decided to understand how controllers and views work by writing some PHP code to display “Hello World” in the home page of the basic template. In order to do this, it is worth understanding what an “action” is. It is simply a function that is always declared inside a Controller class. It is responsible for responding to user requests and sending the final result to the view.

Since I wanted to display “Hello World”, I declared a function called actionSay inside the SiteController class, which extends the Controller class. The function takes in a message parameter, and calls the render() method to create a view file that will display the final message. This is what the function looks like inside the SiteController class:

**<?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]);

}

}

It is worth noting here that if no message is provided in the URL, the default value of $message will be displayed in the browser, which is “Hello” in this case. Secondly, the render method here will display the value of the $message variable in a view file called say.php, which I will have to declare next.

Once the action was created, I had to create a view that would display the result of this action’s execution i.e. the message entered by the user in the URL or the default value. Therefore, I created a new file called say.php in the *views* folder of the basic directory, which essentially displays the message parameter that it receives from the action. This is what it looks like:

**<?php**

**use** **yii**\**helpers**\**Html**;

**?>**

**<?**= Html::encode($message) **?>**

Interestingly, you will note that the message parameter has been HTML-encoded; this has been done to prevent cross-scripting attacks (XSS) since they manipulate Javascript code. This is part of what makes Yii so secure.

Finally, in order to view my message, I simply entered the following URL:

<http://localhost:8080/index.php?r=site%2Fsay&message=Hello+World>

The message parameter includes the message that I want to display (if this was empty, only “Hello” would be displayed) while the r parameter includes the route, which is a unique application-wide ID that has the format ControllerID/ActionID. The ControllerID part is used to determine the controller class that should be accessed, while the ActionID part is used to determine the action that should be executed in that controller class. In this case, the *site/say* route basically says that the SiteController class should be accessed, and the say action should be executed from it.

The next step was to start including models along with the controllers and views, and for this, I decided to create a simple entry form in the home page.

To start with, I created a file called EntryForm.php in the *models* directory, which consists of the EntryForm class that contains the public variables $name and $email, as well as a function called rules that will perform validation on the data entered by the user.

**use** **Yii**;

**use** **yii**\**base**\**Model**;

**class** **EntryForm** **extends** **Model**

{

**public** $name;

**public** $email;

**public** **function** **rules**()

{

**return** [

[['name', 'email'], 'required'],

['email', 'email'],

];

}

}

In the same file, I proceeded to create an EntryForm object on which the validation() method is called. This triggers the data validation routines that need to be performed when inappropriate data is entered.

**<?php**

$model = **new** EntryForm();

$model->name = 'Hamza';

$model->email = 'bad';

**if** ($model->validate()) {

// Good!

} **else** {

// Failure!

// Use $model->getErrors()

}

Next, I created an action for the purpose of displaying the entry form and accepting data from the users. This involved creating the actionEntry function in the SiteController class in which an EntryForm object called $model is created and populated with data from the $\_POST.

**public** **function** **actionEntry**()

{

$model = **new** EntryForm();

**if** ($model->load(Yii::$app->request->post()) && $model->validate()) {

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

} **else** {

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

}

}

}

As can be seen from the code, I rendered two views called entry-confirm and entry that will display the entry form and the user’s responses respectively. Therefore, I created the entry-confirm.php and entry.php files in the *views* directory.

The entry.php file will display the entire EntryForm object that comprises of the name and email text boxes that prompt user input:

**<?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**(); ?>

A widget called ActiveForm has been used here, which essentially creates HTML forms and performs data validation in Yii. Meanwhile, the entry-confirm.php file is merely supposed to display the valid values that have been entered by the user in the form of an unlabeled list.

**<?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>

**Plugins/Extensions**

I deployed two plugins during my time with Yii, including YIMP (a control panel built on Bootstrap 4) and DatePicker (a date/time picker widget for Yii). Both of them were installed via Composer in the CMD terminal, in this way for datepicker and yimp respectively:

php composer require nex/yii2-datepicker

php composer require dmitrybtn/yii2-yimp:dev-master