**SYMFONY**

PART I

**Intro**

Symfony is a PHP web application framework and a set of reuseable PHP components/libraries.

File structure

You will mostly be working in the app and src folders, and occasionally the web folder.

The bin, var and vendor folders are reserved mostly for the core symphony tools and other libraries.

The app folder is used for config files and views:

Config -> routes and db configuration

Resources -> views and templates

The src folder will hold controllers and models:

Appbundle -> mainly used folder – where controllers are stored

The web folder holds the public assets we will use in the app, e.g. css, js, plugins, images etc…

**Installation**

Command line symfony installation through composer on official website. Make sure to set permissions on folder once created in order to view in browser

**Terminal commands**

|  |  |
| --- | --- |
| **Description** | **Cmd** |
| Start the server | Php bin/console server:run |
| Clear cache | Php bin/console cache:clear |
| Create the DB | Php bin/console doctrine:schema:create |
| Validate the DB | Php bin/console doctrine:schema:validate |
| Update the DB | Php bin/console doctrine:schema:update –force |
| Start the wizard to create an entity | Php bin/console doctrine:generate:entity |
| Create an entity | Php bin/console doctrine:generate:entities AppBundle/Entity/***Name*** |
| Generate missing getters and setters | Php bin/console doctrine:generate:entities AppBundle |

*It is good practice to clear the cache every now and again*

**Controllers**

Controllers are responsible for loading information data and passing it into the view. Controllers manage the data from the model, usually data from the database, this data is then sent to the view as needed.

The MVC structure of symfony keeps the code cleaner, more readable and easier to maintain, along with keeping it modular.

The controller handles request and response (HTTP requests e.g. form data, url parameters and other headers). The response alters the behaviour of the browser, e.g. render HTML, respond to an API format, redirect to another method/controller, show error, force download, etc…

**Routes**

Routes are the url entry points of an application. They define what and where the application should call when the user clicks on a link or enters a page in the url. Routes can also be defined as; a map from the defined url to a controller action (method).

Default routing for symfony is done in *annotation*, see below:

/\*\*

\* @Route(“/”. Name=”home”)

\*\*/

The method is then written under this route. Symfony will return an error if nothing is returned from the method. Symfony will automatically search *app>resrouces>views* for a view.

It is good practice to have a folder for each controller view, and a file for each method view:

App>resources>views>ControllerName>index.html.twig //where index is name of method

**Send parameters to a controller**

/\*\*

\* @Route(‘/reservations’, name=’reservations’)

\*\*/

Public function reservations() {

$value = [];

Return $this->render(‘admin/reservations.html.twig’, [‘value’ => $value]);

}

To access the variable in the view, use {{ value }}

Reservations.html.twig

//when linking in your webpages, use the path variable

<a href= {{ path(‘reservations’) }} //where reservations is the name of the route

//to pass parameters

<a href = {{ path(‘reservations’, {‘id’:1}) }}

//the route in the controller should be:

@Route(‘/reservations/{$id}’, name=’reservation\_update’);

//The method should then have the parameters as an argument, along with the *Request $request* parameter.

**Views**

Views are what the user will see rendered on the browser. Views should not contain much logic.

//example of for loop in twig (if array passed from controller):

{% for client in clients %} //where clients is the variable passed

<h2>{{ client.name }}</h2>

{% endfor %}

**Creating templates with twig**

Blocks define sections of our view:

{% block body %}

{% endblock %}

Use assets in the head section to link scripts and stylesheets:

<script src=”{{ asset(‘js/app.js’) }}” ></script>

If you wish to add a specific stylesheet or script for a specific view, you can do so using a block from the template and accessing it in the page specific view:

*Template*

{% block stylesheet %}{% endblock %}

*View*

{% block stylesheet %}

<link rel=’stylesheet’ href=’{{ asset(‘css/view.css’) }}’ />

{% endblock %}

To use a template:

{% extends ‘templateName.html.twig’ %} //at top of view

Page specific mark up goes between the blocks:

{% block body %}

<!—markup here -->

{% endblock %}

**Request**

The *Request $request* is used to grab the data from a form.

**Forms**

Inside a controller, at the top of the script, make sure to add form specific types, as so:

Use Symfony\Component\Form\Extension\Core\Type\*TextType*;

|  |  |  |
| --- | --- | --- |
| TextType | TextareaType | EmailType |
| IntegerType | PasswordType | ChoiceType |
| EntityType | DateType | TimeType |
| DateTimeType | CheckboxType | RadioType |
| FileType | HiddenType | SubmitType |

In method

//dummy entity data below

$task = new Task();

$task->setTask(‘write a blog post’);

$task->setDueDate(new \DateTime(‘tomorrow’));

$form = $this ->createFormBuilder($task)

->add(‘task’, TextType::class)

->add(‘dueDate’, DateType::class)

->add(‘save’, SubmitType::class, array(‘label’ => ‘Create Task’))

->getForm();

Return $this->render(‘/default/new.html.twig’, array( ‘form’ => $form->createView() ))

In view

{{ form\_start(form) }}

{{ form\_widget(form) }}

{{ form\_end(form) }}

By default, the form will submit a POST request back to the same controller that renders it.

Handling form submission

*Example:*

Public function new(Request $request) {

$task = new Task();

$form = $this ->createFormBuilder($task)

->add(‘task’, TextType::class)

->add(‘dueDate’, DateType::class)

->add(‘save’, SubmitType::class, array( ‘label’ => ‘Create Task’ ))

->getForm();

$form->handleRequest($request);

If($form->isSubmitted() && $form->isValid()) {

$task = $form->getData();

//now perform action such as saving data to database

//redirect if form successfully submitted

Return $this->redirectToRoute(‘route\_name’);

}

//show form again if error with form

Return $this->render(‘default/new.html.twig’, array(‘form’ => $form->createView() ))

}

**Intro to Doctrine**

Doctrine is a database handler, it is the home to several PHP libraries, primarily focused on database storage and object mapping. Doctrine makes database interaction easier than using SQL queries. It allows us to extract the database data in objects and clases.

Doctrine is an ORM (object-relational mapper) – this writes queries for us.

Doctrine is used to create models in symfony. A model is simply an extraction from a database table. The model is used merely as a getter and setter container for the DB object.

**Entity class**

An entity class is like a database table. It will create a model in *src>appbundle>entity* with column names and setter & getter methods. The actual creation of the table is done so through the command line. See below.

**Create the application models**

App>config>parameters.yml //this is where the DB credentials go

In terminal

Php bin/console doctrine:database:create //creates database

Php bin/console doctrine:generate:entity //This will throw up a wizard in the terminal

*Wizard:*

Entity shortcut name: AppBundle:Name //where Name is table name

Configuration format: yml

New field name: columnName

Field type: string //or other data type

fieldLength: 20 //or other value

is nullable: false //or true

unique: false //or true

The model will now be generated. This **does not** create the table in the database.

Php bin/console doctrine:schema:update –force //to create table in DB

**Database relationships**

Src>AppBundle>resources>config>doctrine //where doctrine config files are

*\*\* See docs for more information \*\**

Once relationship has been created:

Php bin/console doctrine:generate:entities AppBundle

Php bin/console doctrine:schema:update –force

**Insert data with doctrine**

At the top of the controller, add the entity class:

Use AppBundle\Entity\NameOfEntity;

In method

If($form->isSubmitted()) {

$form\_data = $form->getData();

$em = $this->getDoctrine() -> getManager();

$client = new Client(); //example of model class

$client->setTitle($form\_data[‘firstName’]); //do for all table fields

$em->persist($client); //prepare data

$em->flush(); //create query (insert data)

**Select data with Doctrine**

In controller

$clients = $this->getDoctrine()->getRepository(‘AppBundle:Client’)->findAll();

//where Client is the name of the model.

//use ->find($id) for row specific data

//select data in controller like:

$name = $clients->getName(); //where getName() is a getter in the model

This select data can also be passed in to the view:

$blog = $this->getDoctrine()->getRepository(‘AppBundle:blog’)->findAll();

Return $this->render(‘blog/index.html.twig’, array(‘blog’ => $blog));

**Edit entities**

$client = $this->getDoctrine()->getRepository(‘AppBundle:Client’)->find($id);

$client->setName($form\_data[‘name’]); //setName is method in model

$em = $this->getDoctrine()->getManager();

$em->flush();

//now redirect user

**Work with relational data**

In order to isolate, revise and test complex queries, it is good practice to create a custom repository class for your entity. Methods containing your query logic can then be stored in this class.

\*\* See docs for more information on this \*\*

**Quiz**

Part 1

1. How do you create a new Symfony app?
2. Explain the following directories folders: App, Src, Web, Vendor & Bin
3. Using the terminal: start the server, clear the cache
4. How do you create routes using *annotation*
5. Return a view from the controller
6. How do you pass values from the controller to the view – give example
7. In the view, create a link which links to another route/page
8. Pass a parameter from the view to the controller

Part 2

1. Create a twig template
2. Create a view which uses the above template
3. Pass an array from the controller in to the view and loop over the data in the view
4. How do you link to stylesheets and scripts in the view
5. What is *Request $request*
6. List the basic form types used in Symfony
7. How can you use form types in the controller?
8. Create a form in the controller to be displayed in the view
9. Handle the form data and redirect if successful

Part 3

1. What is Doctrine
2. What file do the DB credentials go in to
3. Create a database
4. Create a new entity/model
5. Create a table from the entity class (terminal command)
6. Where are the models stored in the directory?
7. Insert data to DB
8. Select all data from DB and display in view
9. Select one row of data from DB and display in view
10. Edit/update DB values