



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

School of Applied Computing

Student Name:	Peter Davies	Student P number:	P079868
Programme:	BSc Web Development		
Module:	Advanced Web Development	Lecturer:	Mike Dacey
Assignment Title:	AWD Assignment Two		
Hand Out Date:	February 2019	Submission deadline:	05/04/2019

Disclosure:	I declare that this assignment is all my own work and that I have acknowledged all materials used from the published or unpublished works of other people. All references have been duly cited.		
Student's Signature:		Date:	05/04/19

Contents

1. Introduction	1
2. Knowledge & Understanding Skills	2
2.1 MVC Architectural Design Pattern	2
2.2 Page & Front Controllers	3
2.2.1 Page Controllers	3
2.2.2 Front Controllers	6
2.3.3 Conclusion	8
2.3 The Template Pattern	8
2.3.1 Conclusion	10
2.4 Symfony Four	11
2.4.1 Symfony's MVC Approach	11
2.4.2 ORM Tool Support	11
2.4.3 Testing	12
2.4.4 Server-Side Templates	12
2.4.5 REST Web Services	13
3. Practical Skills	14
3.1 Tools & Technologies Used	14
3.1.1 Composer	14
3.1.2 Git Version Control, Azure DevOps Repository & Pipelines	15
3.1.3 PHP Storm	17
3.2 Development with Symfony	17
3.2.1 Routes, Pages & Templates	17
3.2.2 The Database Structure	21
3.2.3 Doctrine Database ORM	22
3.2.4 CRUD Data Using Controllers	26
4. Evaluation	39
5. References	40
6. Bibliography	43
7. Appendix	45

Table of Figures

Figure 1 - Model-View Controller	2
Figure 2 - Page Controller Pattern	4
Figure 3 - ASP.NET Web Forms Page Controllers	4
Figure 4 - Page Controller Inheritance	5
Figure 5 - Front-Controller Pattern	6
Figure 6 - Front-Controller Sequence	7
Figure 7 - Template View Pattern	8
Figure 8 – Symfony Twig Syntax	9

1. Introduction

The following report firstly, in the knowledge and understanding section, investigates the well-known Model View Controller (MVC) design pattern. It then discusses the front and page controller patterns used when implementing the MVC pattern and finally discussing the template pattern, also commonly found when using an MVC framework.

In the practical skills section, the report documents the production of a simple MVC application using the Symfony 4 framework which creates, reads, updates and deletes data from a database using ORM tools and template views. The code for this application accompanies this report or can be cloned from the Azure DevOps repository found [here](#).

To be able to query the database, a local MySQL server, such as WAMP is required. The SQL accompanying this report can be uploaded create the database and populate it with data. To run the internal Symfony server to view the application, the command line interface is required where the directory of the application must be navigated to (will be different on another machine), and the server started using the following command.

```
C:\Users\Peter>cd desktop/gitrepos/AWD  
C:\Users\Peter\Desktop\gitrepos\AWD>php bin/console server:run  
[OK] Server listening on http://127.0.0.1:8000  
// Quit the server with CONTROL-C.
```

The application can then be viewed in a web browser by navigating the address displayed.

2. Knowledge & Understanding Skills

2.1 MVC Architectural Design Pattern

The MVC architectural pattern is now a commonly used paradigm in object orientated applications [1]. The pattern itself was first defined in the Smalltalk programming language in the 1970's, the language considered the foundations of a 'new-world' of computing [2], however, the MVC pattern is accredited to Prof. Trygve Reenskaug who wrote the original pattern in 1978 at Xerox PARC [1], [3], [4].

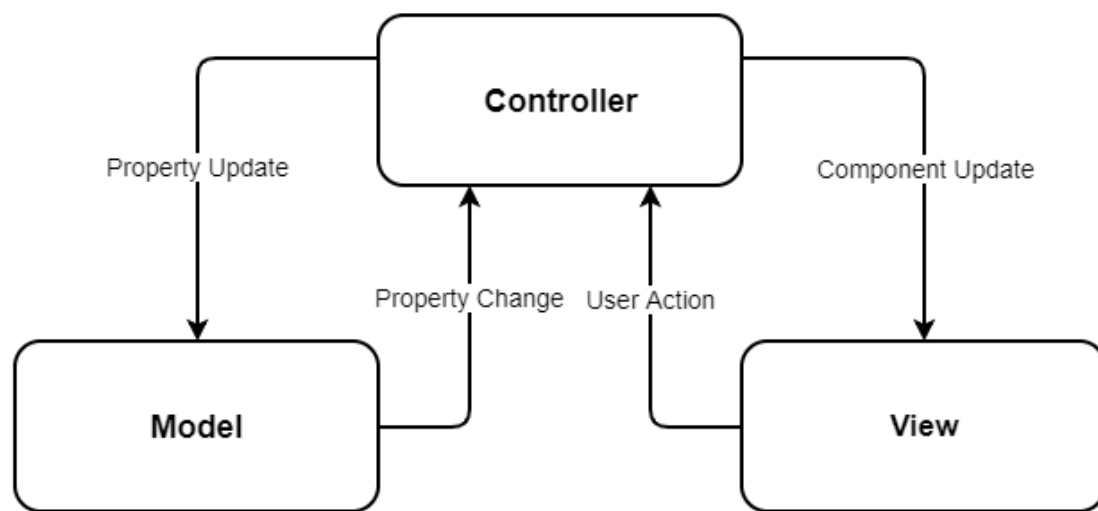


Figure 1 - Model-View Controller

As seen in figure one, the MVC pattern takes into consideration three roles which are separated into different classes [5]:

The Model – the management of the behaviour and data of the application domain, usually that of a database or store [6]. The model is the class which holds information about the data domain, an object within a *domain model* [7] or representations of a relational database table [8]. The model responds to instruction to change the state of the data from the controller and requests from the view.

The View – manages the display of information from the model to the end user of the system [6]. In web applications the view is usually a HTML/PHP or CSHTML (.NET) page [7]. The view

specifies how the model's data will be displayed and will present any updates to the model [3].

The Controller – renders and displays the model's content to the view and the users by handling the users inputs through a mouse or keyboard and outputting the information about the interactions with the view to the model and or the view to change [6]. Where the model does not know of its views and the views know only of their model, the controllers know their views but the views does not know the controller [1].

The overall design principle that MVC offers is a separation of concerns: separating the model from the view and the view from the controller. Fowler [7] states that the separation of view from model is one of the most foundational principles of excellent software design, he continues, adding that the fact the view depends on the model but not vice-versa is key as programmers working on the model can be completely unaware of what presentation is being used, thus allowing for new views to be created and attached to a model without re-creating it, furthermore, multiple views can be attached to a model, with the ability to be presented in different ways [9].

2.2 Page & Front Controllers

The MVC architectural pattern supports two pattern methods for implementation. These are Page controllers and Front controllers. Each of these methods has their own disadvantages and benefits which will be evaluated and discussed in the following sections.

2.2.1 Page Controllers

The page controller pattern is the simpler of the two MVC methods to understand as one page has one controller associated to it, which can either be in the view itself, or a separate object which relates to the view [7].

Figure 2 shows the basic structure for a page-controller MVC application. Each page or view has one controller, or module, on the web server associated with it, controlling the rendering of the items on that page. The user requests on each page are handled directly the class behind the page where they extract the relevant data required, they then invoke updates to

the model, where required, and return the users request view for processing by the browser [10], the view is also reliant on the model for the data it needs to display [11].

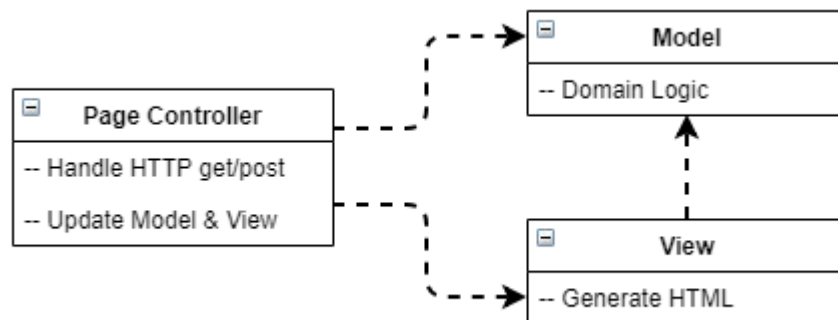


Figure 2 - Page Controller Pattern

A classic example of the page-controller pattern in use is that of a ASP.NET web forms application. Here, each .aspx page has a controller associated with it, for the 'About' page below, the controller is called 'About.aspx.cs'. This pattern is combining the two roles of view and controller.

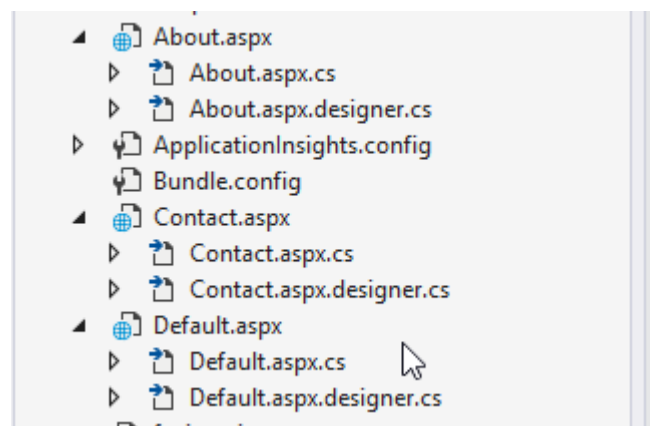


Figure 3 - ASP.NET Web Forms Page Controllers

Fowler [7] adds that page controllers can be implemented in a number of ways; as scripts or as server pages written in ASP, PHP, or JSP, for example. However, creating controllers for each view will result in significant code repetition, therefore, it is usually common practice to create a 'base controller' from which other controllers inherit common functionality, such as validation [11] (figure 4).

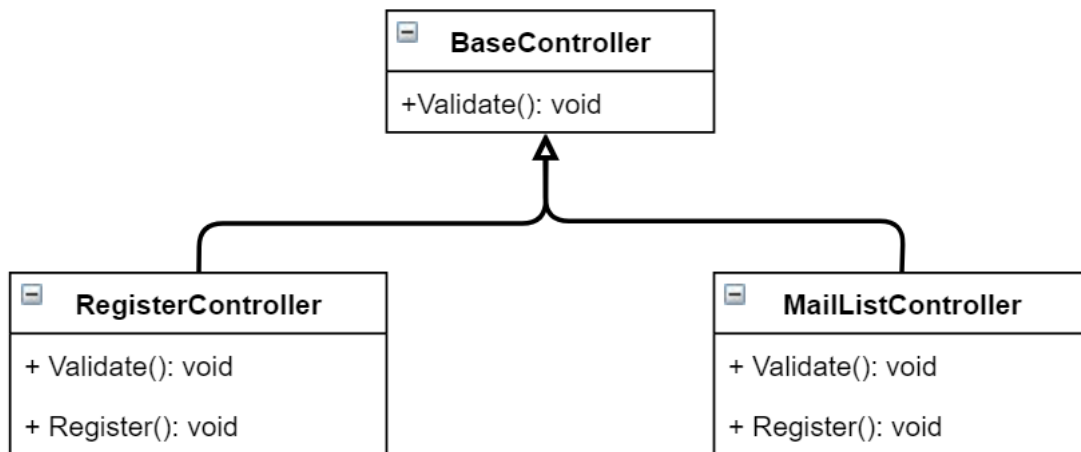


Figure 4 - Page Controller Inheritance

Yet, even with inheritance avoiding repetition of code, the page controller is not considered suitable for larger projects as when the variations of pages grows, the inheritance tree also gets larger, meaning more and more complex applications which creates bottlenecks in the base classes and problems relating to maintenance and support [10].

Microsoft [11] state that modern frameworks have allowed for programmers to easily ‘blend’ together the view and controller related code, thus making it difficult to separate controller logic. This in turn has led to the belief that the page controller pattern is bad design as controller classes can become bloated and testing is becomes difficult.

Although some consider the page-controller sub-standard to the front-controller (section 2.2.2), the page-controller pattern is a perfectly viable option which offers many benefits such as [7][11]:

Simplicity – using a page-controller pattern creates a *natural* file structure which is instantly recognisable and familiar to work on. Furthermore, as each view is handled by one specific controller, each one can remain simple.

Extendable – helper classes/object can be invoked when the controller’s logic complexity increases. This logic can be delegated to other classes to avoid duplication of code.

Frameworks – Most modern web application frameworks, such as ASP.NET, Laravel or Symfony have page controller functionality built in.

2.2.2 Front Controllers

In his book *Pattern for Enterprise Architecture* Fowler [7] states:

“In a complex Web site, there are many similar things you need to do when handling a request. These things include security, internationalization, and providing views for certain users. If the input controller behaviour is scattered across multiple objects, much of this behaviour can end up duplicated. Also, it’s difficult to change behaviour at runtime.”

As discovered in section 2.2.1, when there is repetition of code the page-controller pattern handles this through inheritance, however, for large applications which may deal with some of the requests stated by Fowler, the controllers can become complex, bloated and hard to maintain. To solve this problem, the front-controller is introduced which handles the requests made by a view through a single controller [7][12].

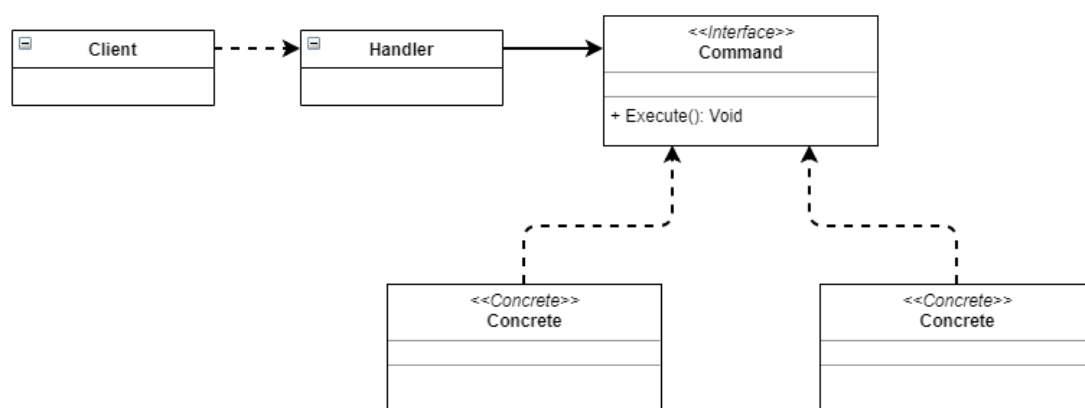


Figure 5 - Front-Controller Pattern

Most MVC applications and frameworks are based on the front-controller design where a central controller resides at the front of all the pages and renders the views based on the central controllers logic [10]. The front-controller is usually implemented in two parts (figure 5) by a web handler and a command hierarchy [7][12]:

Web Handler – receives HTTP requests, either GET or POST and retrieves the parameters deemed relevant from the request and then chooses which action to initiate and through which command (figure 6).

Command Hierarchy – carries out the actions by receiving parameters from the handler and selecting the correct command to process.

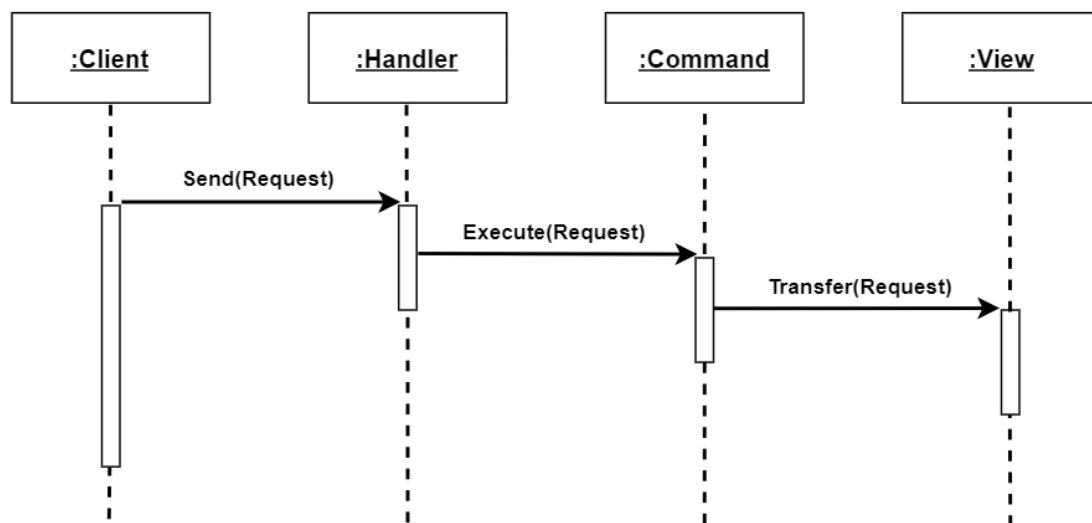


Figure 6 - Front-Controller Sequence

The commands also form part of the controller and are further described in the *GOF's Command* design pattern where controllers are able to interact with commands in a generic way as each command is represented as a single object through an abstract command class (figure 5) which is an interface used for executing the operations of client handler [9].

A good example of the front-controller in operation would be that of navigation. A web application may contain two pages such as *'register.php'* and *'mailinglist-signup.php'* which would each contain similar code to execute commands, by using a front controller all the similar code can be routed through a single controller and the relevant view can be displayed without the need for code duplication.

As with the page-controller there are still flaws with the front-controller, one of which is its complexity. The front-controller is a custom built controller which requires a deeper understanding of the MVC, front-controller, command, decorator and façade patterns [7][12] which are used by modern frameworks.

The front-controller, when understood, does offer many benefits over the page-controller, as you would hope from a more complex solution. These benefits are [7][12]:

Code Repetition – as code is all held in one centralised controller, it is possible to avoid duplication of code that would otherwise be found when using the page-controller.

Thread-Safe – new command objects are created with each request from the view, therefore the complexities of multi-threaded programming can be avoided.

Configuration – only a singular front-controller must be created and configured on the web server meaning a simplified server configuration process is possible.

2.3.3 Conclusion

Both MVC design methods offer benefits and drawbacks as laid out in the preceding sections and decisions on which one to use for a web application will hinge on what requirements need to be met. Fowler [7] adds that it is it isn't uncommon for both methods to be used in web applications, adding that the two mix well together, especially if refactoring from one another.

2.3 The Template Pattern

The template view pattern extracts information to HTML through embedded markers in each HTML page. Each marker is used as a placeholder to display dynamic content on the page in relation to the marker. The content could be data from a database or a page section, such as a header or footer [7].

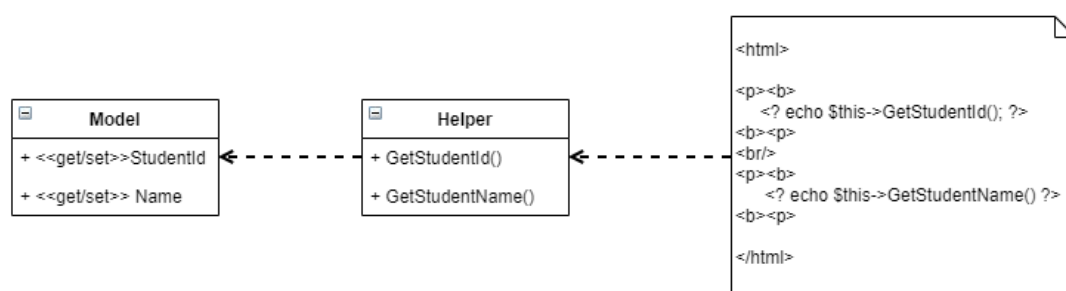


Figure 7 - Template View Pattern

Figure 7 shows a very simple example of how a template view works. A model class represents data stored in a database for a student's ID and name, a controller or helper class contains the logic used to retrieve the ID and name from the database, then within the HTML page, the

helper methods are called, in a mini-php script which echoes out the results and is rendered onto the HTML page or view. A simple way of understanding the template view is by recognising its similarities with php 'includes', where static pages can be broken up into separate php pages to avoid repeating the same code, like a navigation menu, which would be implemented on every page of a web application.

Furthermore, more complex examples of the template pattern in use are found in MVC frameworks such as Symfony and ASP.NET where built in helpers allow for the use of programming code directly in a HTML page, for example, Symfony uses a template engine known as 'Twig' [13] which uses special syntax as markers for displaying dynamic content.

```
<!DOCTYPE html>
<html>
  <head>
    <title>AWD Twig Example</title>
  </head>
  <body>
    <h1>{{ page_title }}</h1>

    <ul id="studentInfo">
      {% for item in studentInfo %}
        <li><p>"{{ item.studentId }}">{{ item.studentName }}</a></li>
      {% endfor %}
    </ul>
  </body>
</html>
```

Figure 8 – Symfony Twig Syntax

Figure 8 shows the special syntax used as markers, where {{ ... }} prints a variable or result of an expression, {% ... %} controls any logic, such as for-loops, for-each-loops or switch statements and {# ... #} allows for comments. This special kind of syntax is not exclusive to Symfony, many other frameworks utilise it, such as Razor pages in ASP.NET where they are known as 'Razor' helpers which are supported through the use of the '@' symbol to transition between HTML and Razor mark-up, again, for-loops, for-each etc. are all supported as are calling of built in functions like '@Datetime' [14].

2.3.1 Conclusion

Fowler [7] states that the template view's strengths are in its ability to allow the developer to compose content on the page by using the page structure, which he adds, helps people learn through its ease of use. Furthermore, it is understood that by using templates to introduce certain sections of a page into a view then it enables a developer to better organise code using '*partials*' (to borrow a phrase from .NET MVC), which can be filed better and avoid having to repeat chunks of code of similar pages.

Fowler [7] does however point to two disadvantages of template views, firstly and most importantly, they allow for complex logic to be written directly into a HTML page, which, although sounds simple, it can cause problems for inexperienced programmers who may see it as a 'simple, quick solution' to an otherwise complex code structure of classes, however, in the long-term this makes code disorganised and difficult to maintain and follow. Furthermore, he adds that testing of code inside template views is more difficult, if not impossible as it is implemented on a web server.

Template views offer a flexible, easy-to-learn, powerful addition to any MVC application which has powerful benefits when used correctly. The use of the template pattern will be demonstrated in part two of this assignment, where the Symfony framework has been used to demonstrate an understanding of the MVC architectural pattern in PHP using a modern framework, which can be found in section 3.

2.4 *Symfony Four*

Symfony is a leading, open-source, PHP framework used to create websites and web applications which are built on top of a library of PHP components [15]. It was created in 2005, under an MIT license by SensioLabs [16].

The following section of this report discusses the approach taken by Symfony and, some of its features.

2.4.1 *Symfony's MVC Approach*

Symfony is based on the MVC design pattern [17], and utilises both of the controller methods that Fowler discusses (sections two): front and page controllers. However, Symfony chooses to call the page controller an '*Action*' controller, but it serves the same purpose; it is a controller which contains code unique to a view [17].

The first entry point into any Symfony application is the front-controller [18], which, in Symfony four is the *index.php* file. Every request is routed through this file, therefore it can be used as a global initializer [19]. The front-controller in Symfony is used to create instances of the Kernel class, the core of Symfony, which will handle the request passed to it and return a response to the browser [19].

2.4.2 *ORM Tool Support*

Object relational mapping (ORM) is a methodology for converting incompatible data type systems with the use of object-orientated languages and programming techniques [20]. It essentially uses the programming language within the program, to talk to the language of the database, usually SQL, so that direct interaction with SQL is with same language the program is developed; in the case of Symfony, PHP.

Symfony comes with the option of two third-party ORM tools, Propel [21] and Doctrine [22], which allow access to data in a database through objects, without the need to query the database explicitly. Using an ORM offers a greater reusability of methods which query a database throughout the application and even from different applications [22].

Many other frameworks also offer ORM tools, such as .NET's Entity Framework [22], however, there are pros and cons of using an ORM. As ORM's hide SQL code, it adds another layer of security against SQL injection attacks, furthermore, it executes quicker and avoids the need to type out lengthy and complex queries as simpler code replaces it. However, some argue that learning ORM is difficult and initial configuration can be complex [23], [24].

After using Doctrine for as the ORM tool for the application, it can be argued that it is one of the simpler ORM tools to use with most of the commands executed through the CLI which creates most of the classes required automatically through an easy to use question and answer system which adds the database entities and their columns with ease, also allowing for column types to be specified.

In comparison with another ORM tool used in the past, .NET's entity framework, it is felt that Doctrine is easier to set up, simpler to use and mistakes are certainly easier to rectify. The documentation Symfony provide for Doctrine also proves very helpful.

2.4.3 Testing

Symfony offers another third party library called PHPUnit [25] which offers a framework for testing.

2.4.4 Server-Side Templates

As discussed in section two, Symfony offers a server side template engine called Twig [25]. When used, these templates compile into plain PHP code. Twig themselves argue that PHP itself is a template engine, as was highlighted in a comparison in section two, however, they add that modern template engines surpass what can be offered by the language itself [26].

Having used Twig templates in section three it could be argued that they are more complex than simple PHP includes and do not offer as much flexibility. The form rendering controls are especially tricky to use, however, given time constraints, proper evaluation and use of these Symfony 'shortcuts' hasn't been carried out, and with practice these features could really aid in development.

2.4.5 REST Web Services

REST or Representational State Transfer was introduced by Fielding in 2000 [27]. Feilding *et al.* [28] state that REST is:

“REST is a coordinated set of architectural constraints that attempts to minimize latency and network communication while at the same time maximizing the independence and scalability of component implementations.”

REST services fetch and query data from a webservice by establishing a one-to-one CRUD mapping using HTTP methods GET, PUT, POST & DELETE, meaning that REST offers a flexible approach independent of platforms or languages [29]. Japikse *et al.* [30] argue that REST is essentially the MVC framework without the view, as it separates the interface from the data storage like MVC. This means that responses from web servers can be given in any format, JSON, XML etc. and an interface can be designed to consume the response, each with their own different views.

Symfony offers a bundle package *FOSRestBundle* [31] which installs into the *AppKernel* file and accepts the clients requests and return the appropriate response. Another bundle, *JMSSerializerBundle* is also required to serialise the data into the requested format (JSON, XML etc.).

3. Practical Skills

The following section of the report documents the construction of an MVC PHP web application using the latest Symfony 4 framework.

The following web app has been created using guidance from Symfony's web documentation [32]. Section three of the report and the development have been conducted prior to writing section 2 to allow for a greater understanding of the Symfony 4 framework.

3.1 Tools & Technologies Used

3.1.1 Composer

Composer is dependency management tool required by many PHP frameworks to declare libraries which each project will depend on, furthermore, it acts as a management and updating tool for the project dependencies [33].

To first create a Symfony, a skeleton template is created which contains the bare minimum of dependencies required to develop into a full Symfony web application. This is done through the command line, using composer.

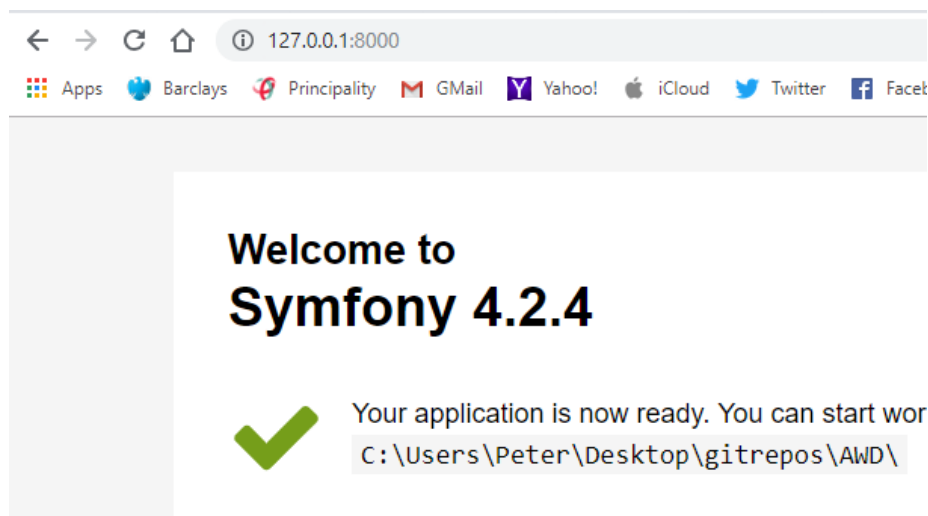
```
C:\Users\Peter\Desktop>cd gitrepos

C:\Users\Peter\Desktop\gitrepos>composer create-project symfony/skeleton AWD
Installing symfony/skeleton (v4.2.4.1)
  - Installing symfony/skeleton (v4.2.4.1): Loading from cache
Created project in AWD
Loading composer repositories with package information
Installing dependencies (including require-dev) from lock file
Package operations: 22 installs, 0 updates, 0 removals
  - Installing symfony/flex (v1.2.0): Downloading (100%)

Prefetching 21 packages
  - Downloading (100%)

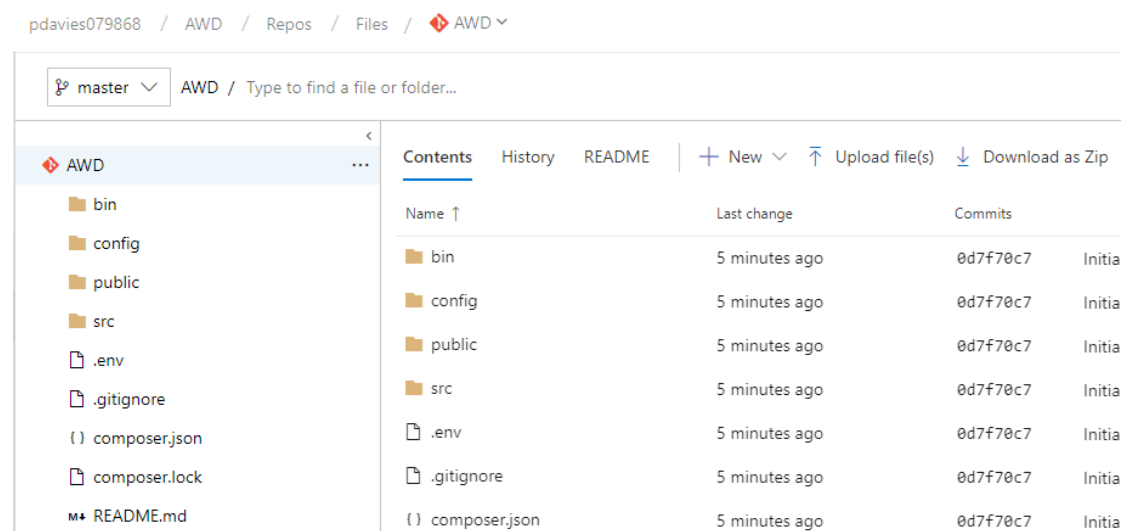
  - Installing psr/cache (1.0.1): Loading from cache
  - Installing psr/container (1.0.0): Loading from cache
  - Installing psr/simple-cache (1.0.1): Loading from cache
  - Installing symfony/polyfill-mbstring (v1.10.0): Loading from cache
  - Installing symfony/polyfill-php73 (v1.9.0): Loading from cache
```

A local development server is then installed using the `composer require server -dev` command which allows the web application to run locally. The server is then started using the `php -S 127.0.0.1:8000 -t public` command and the web app is run in the browser.



3.1.2 Git Version Control, Azure DevOps Repository & Pipelines

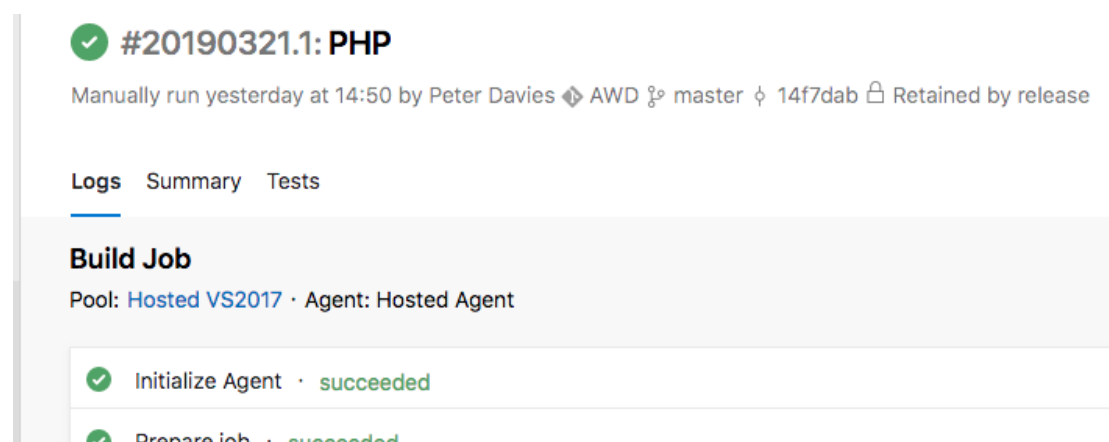
When creating the symphony application, a git [34] repository was created in the project folder using the `git init` command to keep track on changes to any files in the application. An online git repository was then created in Azure DevOps [35], called AWD, which allows for any code to pushed/pulled to and from the remote web based repository which will be cloneable from anywhere, permitting development across several computers, at any desired location.



It is worth noting that repositories of this type are designed for serving teams of developers, each working on different sections of an application, however, it is considered professional

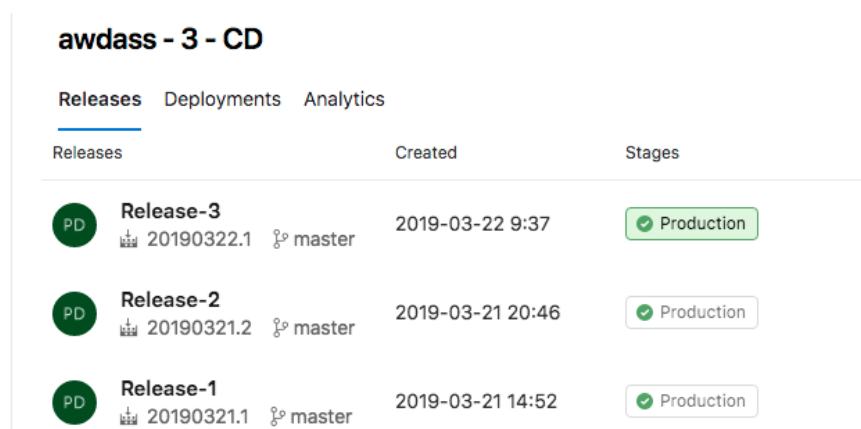
practice for developers worldwide and therefore has been used for this assignment. Lecturer Michael Dacey has been added to the DevOps AWD team to allow access to code for review, if required.

A separate client is used for managing the git process, called GitKraken [36]. It is available for free download with limited features, however, a full-feature version is being utilised through the GitHub student developer pack [37]. Furthermore, an Azure DevOps Pipeline [38] has been created to automate the build and deployment process.

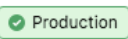
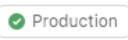
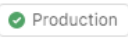


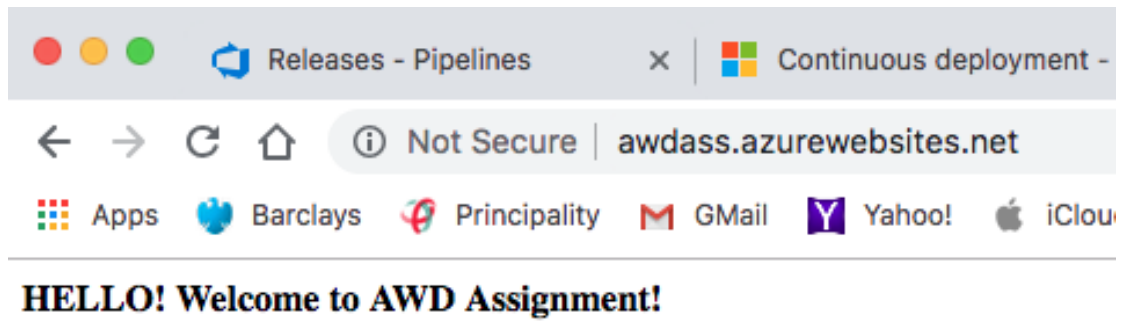
The screenshot shows a build job titled "#20190321.1: PHP" with a green checkmark icon. Below the title, it says "Manually run yesterday at 14:50 by Peter Davies" followed by icons for AWD, master branch, and commit 14f7dab, and a note "Retained by release". There are tabs for "Logs", "Summary", and "Tests", with "Summary" being the active tab. Under "Summary", the section "Build Job" is shown, with "Pool: Hosted VS2017" and "Agent: Hosted Agent". Below this, two steps are listed: "Initialize Agent" and "Prepare job", both with green checkmarks and the status "succeeded".

When using a pipeline, the repository can be pushed locally to Azure DevOps to undergo the build process which tests the code automatically (shown above). The build is then deployed directly to the azure portal server automatically. The website is then live can then be viewed at <http://awdass.azurewebsites.net>.



The screenshot shows the "Releases" page for a project named "awdass - 3 - CD". There are tabs for "Releases", "Deployments", and "Analytics", with "Releases" being the active tab. The page displays a table of releases with columns for "Releases", "Created", and "Stages".

Releases	Created	Stages
 Release-3  20190322.1  master	2019-03-22 9:37	 Production
 Release-2  20190321.2  master	2019-03-21 20:46	 Production
 Release-1  20190321.1  master	2019-03-21 14:52	 Production



3.1.3 PHP Storm

PHP Storm [39], by JetBrains is an integrated development environment designed specifically for PHP. It has been chosen for development as it is available across all platforms, has git integration, provides error prevention, automatic refactoring, automatic code completion and has a vast range of Symfony plugins to help with the development.

3.2 Development with Symfony

3.2.1 Routes, Pages & Templates

The first step in developing with Symfony was installing the skeleton project as discussed in section 3.1.1. Once the framework was installed, the next step was to create pages and routes to those pages, using controllers, where the route defines a URL to the page and the controller is a function which return the content for that page.

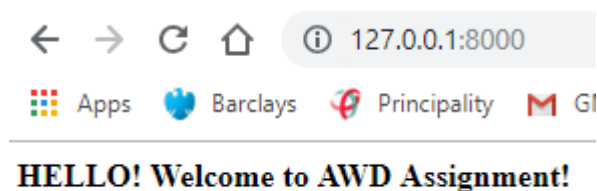
Firstly, a controller is created called '*HomeController.php*' which will contain the routing controller methods to return responses depending on what URL is specified. By default, Symfony uses the *route.yaml* file to route to a controller and its method for returning response objects.

```
index:
  path: /
  controller: App\Controller\HomeController::homepage
```

This route file tells the front-controller (index.php) to use the *HomeController* and call the *HomePage()* method to return a response. These methods are known as *Actions* in Symfony, but they are really just page-controllers as explained in section 2.4.1.

```
public function HomePage()  
{  
    return new Response( content: '<b>HELLO! Welcome to AWD Assignment!</b>');  
}
```

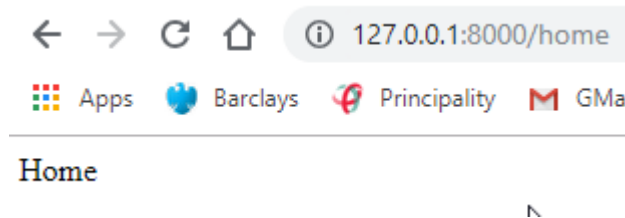
When navigating to the local host, this results in the following page being displayed.



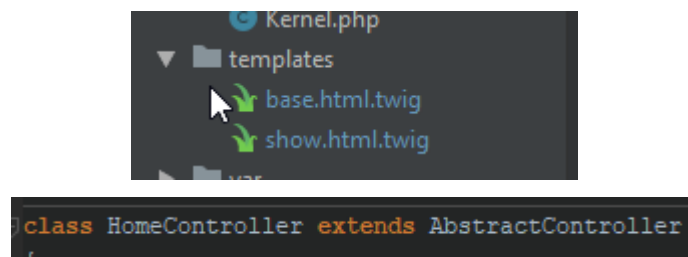
However, Symfony offers a simpler way of routing to different pages; *annotations*. Annotations allow developers to specify the routes and returned response for each URL directly in the controller class. Annotations are installed using the command line and composer, the *routes.yaml* code is commented out, then the following syntax is used to specify URL routes in the *HomeController* class.

```
/**  
 * @Route("/")  
 */  
public function HomePage()  
{  
    return new Response( content: '<b>HELLO! Welcome to AWD Assignment!</b>');  
}  
  
/**  
 * @Route("/home")  
 */  
public function Show()  
{  
    return new Response( content: 'Home');  
}
```

Now, the route for the index of the application is specified by `@Route("/")`, which results in the same page being displayed earlier. The second response method shown `Show()` has a route of `("/home")` meaning that when visiting <http://127.0.0.1:8000/home> a different response is returned, this time, just 'Home'.



Currently, these actions just return plain HTML objects hard coded, however, for a working application, full HTML pages will need to be returned. In Symfony, these pages, or views as per MVC, are called *Twig Templates*. To utilise these server-side templates, composer is required to install the Twig dependencies which installs a Symfony *Bundle* which are plugins to add functionality to the application. The twig templates, or views, are stored in the templates folder at the root of the directory.



To enable the *HomeController* class's actions to return a view, the class must extend the *AbstractController* class which contains the *Render()* method which allows the view to be returned. A action method can then be created which returns the specified view, in this case, the default '*base.html.twig*' which has been modified to demonstrate the page being rendered. The URL has been routed to `("/home1")`.

```
/**
 * @Route("/home1")
 */
public function ShowTwig()
{
    return $this->render( view: 'base'); // Return a HTML view
}
```

```
php x Kernel.php x routes.yaml x HomeController.php x styles.css x base.html.twig x
<!DOCTYPE html>
<html>
  <head>
    {% block stylesheets %}
      <link href="{{ asset('assets/css/grid-fluid.css') }}" rel="stylesheet"/>
      <link href="{{ asset('assets/css/styles.css') }}" rel="stylesheet"/>
    {% endblock %}
    <meta charset="UTF-8">
    <title>{% block title %}Welcome!{% endblock %}</title>
  </head>
  <body>
    <p>My first AWD page using a route!</p>
  </body>
</html>
```

CSS files have also be added to the page using Symfony’s *asset* component. When navigating to <http://127.0.0.1:8000/home1> the following view is returned:



My first AWD page using a route!

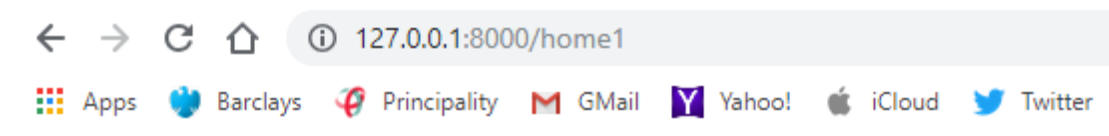
To tidy up the page and demonstrate the full potential of *twig* templates, the main content currently on the page will be moved to another *twig* file called ‘*show.html.twig*’, then, using the `{% block body %}` `{% endblock %}` tags wrapped around the body content it is possible to render this content in between the corresponding tags in the ‘*base.html.twig*’ file.

```
{% extends 'base.html.twig' %}

{% block body %}
  <h1></h1>
<div>
  <p>My First Page using a block</p>
</div>
{% endblock %}
```

```
</head>
<body>
  {% block body %}{% endblock %}
</body>
</html>
```

When the page is refreshed, the new content in the *show* twig file is rendered into the *base* template and displayed on the page:

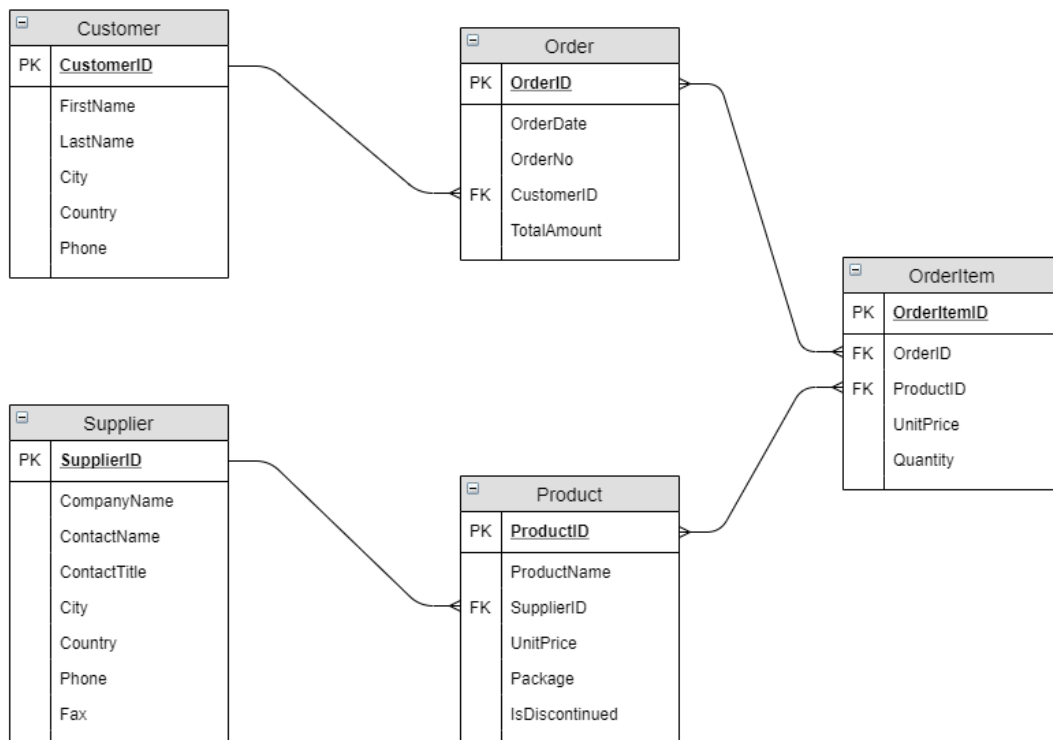


My First Page using a block

This template engine, as previously mentioned is much like a *PHP include*, however, it supports many more functions such as for-loops and foreach loops etc. Now that Symfony routes, controllers and templates have been investigated and tested, they will be used as the basis to form the rest of the application.

3.2.2 The Database Structure

Given the time constraints on this assignment, the database used for this assignment has been created from test SQL files from DoFactory's tutorial [40]. The database is a simple inventory management system holding information about customers, products, orders and suppliers and is supplied with test data. The following ERD explains the relationships between the tables. The SQL files will accompany this report.



3.2.3 Doctrine Database ORM

To create the database and mapping to the database, Doctrine has been installed using composer. Doctrine is Symfony's third party ORM tool, as discussed in section 2.4.2.

```

C:\Users\Peter>cd desktop/gitrepos/AWD

C:\Users\Peter\Desktop\gitrepos\AWD>composer require doctrine
Using version ^1.0 for symfony/orm-pack
./composer.json has been updated
Loading composer repositories with package information
  
```

Once composer has installed all the required dependencies, there are two files for configuring the database. The first one, `.env` is found at the root of the project. In this file the database connection environment variable is set-up:

```

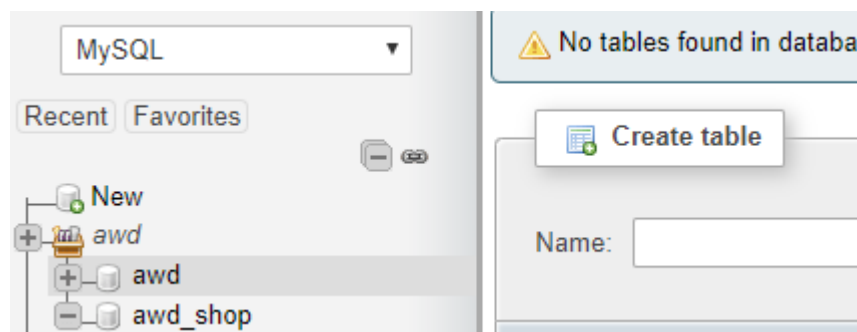
# Configure your db driver and server_version in config
DATABASE_URL=mysql://root:p079868@127.0.0.1:3306/AWD
###< doctrine/doctrine-bundle ###
  
```

This has configured the connection to use a username of 'root' with password 'p079868', connecting to a database called 'awd_shop' on port 3306, the SQL development server installed by WAMP.

The database can then be created, using composer and doctrine using the following command:

```
C:\Users\Peter\Desktop\gitrepos\AWD>php bin/console doctrine:database:create
Created database 'awd_shop' for connection named default
```

Doctrine has now created and SQL database in phpMyAdmin on the mySQL local server:



To create the entities which will be mapped using doctrine another Symfony bundle is installed called *Maker* which will allow for the php class entities to be created using the command line and mapped using doctrine. Again, this is installed using composer.

```
C:\Users\Peter\Desktop\gitrepos\AWD>composer require symfony/maker-bundle --dev
Using version ^1.11 for symfony/maker-bundle
./composer.json has been updated
Loading composer repositories with package information
Updating dependencies (including require-dev)
```

Maker is then used to create the PHP class entities for the database: Customer, Order, OrderItem, Product and Supplier. The first table Customer and the first field is shown below.

```

C:\Users\Peter\Desktop\gitrepos\AWD>php bin/console make:entity

Class name of the entity to create or update (e.g. FierceKangaroo):
> Customer

created: src/Entity/Customer.php
created: src/Repository/CustomerRepository.php

Entity generated! Now let's add some fields!
You can always add more fields later manually or by re-running this command.

New property name (press <return> to stop adding fields):
> CustomerID

Field type (enter ? to see all types) [string]:
> integer

Can this field be null in the database (nullable) (yes/no) [no]:
> no

updated: src/Entity/Customer.php

Add another property? Enter the property name (or press <return> to stop adding fields):
> _

```

Once all the fields have been added, an entity class has been created in the entity folder found in src directory in the project. As the properties are private, GET/SET methods are also added.

```

/**
 * @ORM\Entity(repositoryClass="App\Repository\CustomerRepository")
 */
class Customer
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;
}

```

An additional *CustomerRepository* class has also been created.

```

/**
 * class CustomerRepository extends ServiceEntityRepository
 * {
 *     public function __construct(RegistryInterface $registry)
 *     {
 *         parent::__construct($registry, Customer::class);
 *     }
 *
 *     // /**
 *     //  * @return Customer[] Returns an array of Customer objects
 *     //  */
 *     /**
 *     public function findByExampleField($value)
 *     {
 *         // ...
 *     }
 * }

```

The process of creating the entities was repeated for each of the tables which has created all the classes which will relate to the business logic (the database) of the application. These classes are considered the *Models* of the application, as per the MVC pattern.

Once the models were created, a migration to create the table in the database was then created using the following command:

```
C:\Users\Peter\Desktop\gitrepos\AWD>php bin/console make:migration

Success!

Next: Review the new migration "src/Migrations/Version20190324155944.php"
Then: Run the migration with php bin/console doctrine:migrations:migrate
See https://symfony.com/doc/current/bundles/DoctrineMigrationsBundle/index.html
```

This creates a migration file in the Migrations directory which contains all the SQL code to create the tables in the awd_shop database.

```
public function up(Schema $schema) : void
{
    // this up() migration is auto-generated, please modify it to your needs
    $this->abortIf($this->connection->getDatabasePlatform() === 'mysql', 'mysql is not supported');

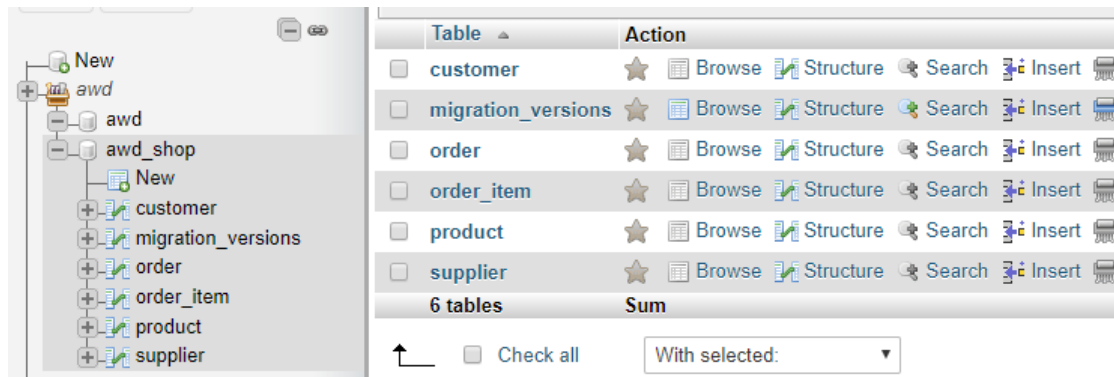
    $this->addSql('CREATE TABLE customer (id INT AUTO INCREMENTAL PRIMARY KEY)');
    $this->addSql('CREATE TABLE `order` (id INT AUTO INCREMENTAL PRIMARY KEY)');
    $this->addSql('CREATE TABLE order_item (id INT AUTO INCREMENTAL PRIMARY KEY)');
    $this->addSql('CREATE TABLE product (id INT AUTO INCREMENTAL PRIMARY KEY)');
    $this->addSql('CREATE TABLE supplier (id INT AUTO INCREMENTAL PRIMARY KEY)');
}
```

The migration file is then migrated using the following command, which runs the SQL in the migration file and creates the tables in the phpMyAdmin database.

```
C:\Users\Peter\Desktop\gitrepos\AWD>php bin/console doctrine:migrations:migrate

Application Migrations

WARNING! You are about to execute a database migration that could result in schema changes.
Do you wish to continue? (y/n)y
Migrating up to 20190324155944 from 0
```



Once all the tables were created, test data was added using SQL in phpMyAdmin. The SQL used to insert test data will accompany the code submission with this report.

Options

				id	first_name	last_name	city	phone
				1	Maria	Anders	Berlin	030-0074321
				2	Ana	Trujillo	México D.F.	(5) 555-4729
				3	Antonio	Moreno	México D.F.	(5) 555-393
				4	Thomas	Hardy	London	(171) 555-77

3.2.4 CRUD Data Using Controllers

Test data has already been added to the database so to demonstrate how data is inserted, read, updated and deleted from the database the following section of the report focuses on querying the *Customer* table.

3.2.4.1 Reading Data

To read data back from the *Customer* table firstly a new controller class is created called *CustomerAdminController* which extends the *AbstractController* class. In this class, a series of use statements are added to firstly add the routes and response classes, but importantly to add the model class *Customer* to allow access to the database properties but also the ORM *EntityManagerInterface* class which allows access to methods in the interface to CRUD the database.

```

namespace App\Controller;

use Doctrine\ORM\EntityManagerInterface;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use App\Entity\Customer;

```

Secondly, a function is added called *Show()* which contains the Symfony code to fetch data. In this function, the first line of code calls access to the *Customer* repository class.

```

/**
 * @Route("/customers")
 */
public function Show(EntityManagerInterface $em)
{
    $repo = $em->getRepository( className: Customer::class);
    /** @var Customer $customer */
    $customer = $repo->findAll();

    // If nothing in database throw exception
    if(!$customer)
    {
        throw $this->createNotFoundException(sprintf( format: 'No records found!'));
    }

    return $this->render( view: 'customers.html.twig', [
        'Customers' => $customer
    ]);
}

```

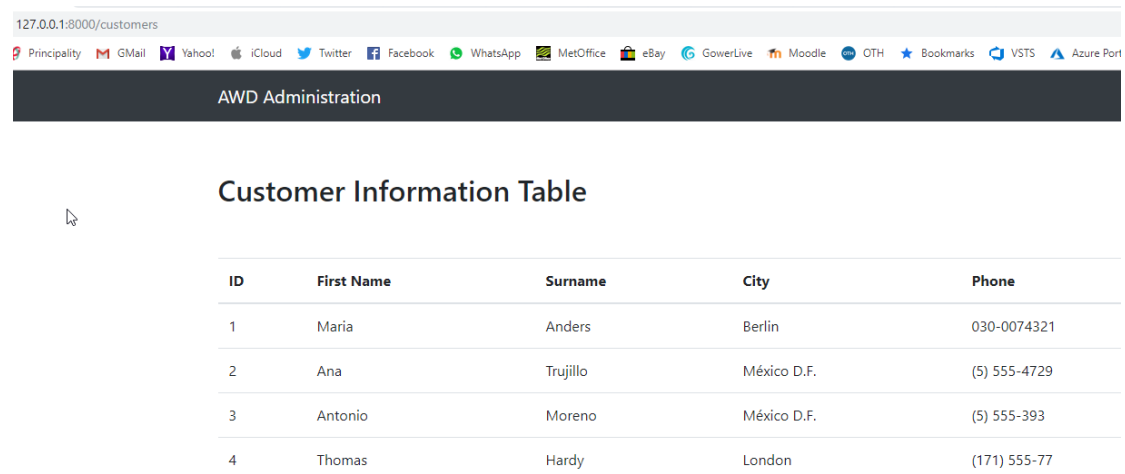
Once access to the repository is stored in the variable *\$repo* and new variable *\$customer* is declared and the method *findAll()* is called on the repository to fetch all the records from the database. Annotations are also given to give a hint to PHP Storm that *\$customer* is a variable of the object *Customer*. A conditional statement is also added to return an exception should there be no records available.

As Symfony controllers must always return a response or object, so the *customers* view is returned but also passed the list of customers returned by the *findAll()* method called earlier. Annotations are added about the method to route to the customers view.

To create a readable interface for the home & customer views bootstrap templates [41][42] were modified and used to save time coding HTML & CSS. Within the customer view, the *twig* template syntax is used to iterate through the contents of the database and calls the *get()* methods inside of the *Customer* entity class for each property, printing each one out.

```
</thead>
<tbody>
{% for customer in Customers %}
<tr>
<td>{{ customer.Id }}</td>
<td>{{ customer.firstName }}</td>
<td>{{ customer.lastName }}</td>
<td>{{ customer.City }}</td>
<td>{{ customer.Phone }}</td>
</tr>
{% endfor %}
</tbody>
</table>
```

Now, when navigating in the browser to the customers page, the database content is printed out in the bootstrap table.



The screenshot shows a web browser window with the address bar displaying '127.0.0.1:8000/customers'. The browser's toolbar includes various icons for social media and services. The main content area has a dark header bar with the text 'AWD Administration'. Below this, the title 'Customer Information Table' is centered. A table with 5 columns (ID, First Name, Surname, City, Phone) and 4 rows of data is displayed. A mouse cursor is visible over the table.

ID	First Name	Surname	City	Phone
1	Maria	Anders	Berlin	030-0074321
2	Ana	Trujillo	México D.F.	(5) 555-4729
3	Antonio	Moreno	México D.F.	(5) 555-393
4	Thomas	Hardy	London	(171) 555-77

3.2.4.2 Inserting Data

To insert data, first a form must be created. In Symfony, form components are used to render forms through special controllers call *FormTypeControllers*. To do this, a new form directory is created in the *src* folder in the project and a new class added called *CustomerFormType* which extends the class *AbstractType* which is installed using composer. Inside this class, the function *buildForm()* is implemented and properties from the *Customer* model added.

```

class CustomerFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add( child: 'FirstName', type: TextType::class, array('attr' => array('class' => 'form-control')))
            ->add( child: 'LastName', type: TextType::class, array('attr' => array('class' => 'form-control')))
            ->add( child: 'City', type: TextType::class, array('attr' => array('class' => 'form-control')))
            ->add( child: 'Phone', type: TextType::class, array('attr' => array('class' => 'form-control')));
    }
}

```

Next, in the *CustomerAdminController* class, inside the function *New()* the *createForm()* method is called, passing in the *CustomerFormType* class which Symfony will use to render the form.

```

public function New(EntityManagerInterface $em, \Symfony\Component\HttpFoundation\Request $request)
{
    $form = $this->createForm( type: CustomerFormType::class);
}

```

Again, as Symfony controllers must return a response object, a template is rendered to the *customerForm.html.twig* template using the *render()* method which inside it another method *createView()* transforms the form object into another object called '*customerForm*' which is returned into the customer form twig view.

```

return $this->render( view: 'customerForm.html.twig', [
    'customerForm' => $form->createView()
]);

```

In the customer form twig page, special twig form rendering functions are used to display the form on the page with the corresponding form name and rows declared in the controller classes.

```

{% block form %}
  <!-- Form -->
  <div class="container">

    <div class="text-center border border-light p-5">
      <p class="h4 mb-4">Add Record</p>
      {{ form_start(customerForm) }}
      <div class="form-row mb-4">
        <div class="col">
          <!-- First name -->
          {{ form_row(customerForm.FirstName) }}
        </div>
        <div class="col">
          <!-- Last name -->
          {{ form_row(customerForm.LastName) }}
        </div>
      </div>
      {{ form_row(customerForm.City) }}
      <small id="spacer" class="form-text text-muted mb-4"></small>
      {{ form_row(customerForm.Phone) }}
      <small id="guidance" class="form-text text-muted mb-4">
        Max 12 Characters! Please remove any spaces.
      </small>
      {{ form_widget(customerForm) }}
      <button class="btn btn-info my-4 btn-block" type="submit">Add Record!</button>
      {{ form_end(customerForm) }}
    </div>
  </div>
  <!-- /Form -->
{% endblock %}

```

Then, using the twig template syntax, this form is placed inside a *block form* tag and rendered into the *base.html.twig* file to be displayed when a user visits the URL route */addCustomer* which is declared in an annotation above the *New()* method in the *CustomerAdminController* class.

AWD Administration
Home

Add Record

First name

Last name

City

Phone

Max 12 Characters! Please remove any spaces.

Add Record!

It is worth noting that a bootstrap template had originally been used for the form, however, the use of the built-in form rendering has placed conflicting styles on the form. Efforts were made to override this using classes added to the *buildForm()* functions properties inside the *CustomerFormType* class, however, the resulting styles are unexpected and are not correct. More research into using Symfony's form rendering templates is needed and the styles evident on the final submission may differ from this report if there is time to research further.

The form rendered on the page contains no 'action' attribute in the <form> tag, meaning that when the form is submitted it will route back to the same URL specified in the *CustomerAdminController* above the *New()* method. This means that the *New()* method renders the form and handles the form request, therefore, the code to insert data lives inside this method.

```
/**
 * @Route("/addCustomer")
 */
public function New(EntityManagerInterface $em, \Symfony\Component\HttpFoundation\Request $request)
{
    $form = $this->createForm( type: CustomerFormType::class);

    $form->handleRequest($request);
    if($form->isSubmitted() && $form->isValid())
    {
        $data = $form->getData();
        $customer = new Customer();
        $customer->setFirstName($data['FirstName'])
            ->setLastName($data['LastName'])
            ->setCity($data['City'])
            ->setPhone($data['Phone']);

        $em->persist($customer);
        $em->flush();

        $this->addFlash( type: 'success', message: 'Record Successfully Added!');

        return $this->redirectToRoute( route: 'app_customeradmin_show');
    }

    return $this->render( views: 'customerForm.html.twig', [
        'customerForm' => $form->createView()
    ]);
}
```

In the above screenshot the HTTP request is passed into the *New()* method, however it is not understood why PHP storm inserts it as the full component address but this was the only way the correct *Request* object could be passed.

The second line of code of interest is the *handleRequest(\$request)* method, this handle's both the GET and POST requests, however, when the page is loaded, to stop the form being submitted this method defaults to a GET request. When the code moves into the *if* block, as

the *handleRequest()* method has returned a GET request, the *isSubmitted()* method returns false and the code skips to the *render()* method which simply displays the form. When the form is submitted using the POST request, *handleRequest()* sees the data in the form (stored in the *\$form* variable) and *\$form->isSubmitted()* returns true, leading to the code inside the *if* block to be executed.

The code inside the *if* block firstly gets the data in the form using the *getData()* method which return an associated array of the fields in the form. Then a new *Customer* object is created using the *new* keyword. The array objects properties are then set using the *get* method from the *Customer* model class with the fields from the form passed in. Saving to the database is handled through the *EntityManager* class which allows the calling of the *persist()* method which is Doctrine's way of saving the data, which is then executed through Doctrine's *flush()* method.

Add Record

First name

Last name

City

SWANSEA

Phone

+447980002273

Max 12 Characters! Please remove any spaces.

Add Record!

5	Christina	Berglund	Luleå	0921-12 3465
22	Peter	Davies	SWANSEA	07980002273

☐ Edit Copy Delete

5 Christina Berglund Luleå 0921-12 3465

☐ Edit Copy Delete

22 Peter Davies SWANSEA 07980002273

The final two lines of code in the *New()* method create a better UX for the insert process. The first method *addFlash()* is Symfony's shortcut for displaying a message, stored in the session, after a successful submit. The second method *redirectToRoute()* simply redirects back to the *customers.html.twig* view which displays the database contents in a table.

Symfony offers another way to insert data into the database, without setting the data to the *Customer* object manually by specifying each field. This is called class binding when the *Customer* model class is bound to the form.

To do this, a new function called *configureOptions()* is created in the *CustomerFormType* class.

```
public function configureOptions(OptionsResolver $resolver)
{
    parent::configureOptions($resolver); // TODO: Change the autogenerated stub
}
```

PHP storm also automatically adds the dependencies required; *OptionsResolver*.

```
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\OptionsResolver\OptionsResolver;
```

Within this method the function *setDefault()* is called with an array specifying the option 'data_class' which is set to the class or model to be bound to the form. In this example, this will be the *Customer* model class. When the form submits, this method will automatically create a *\$customer* object.

```
public function configureOptions(OptionsResolver $resolver)
{
    $resolver->setDefaults([
        'data_class' => Customer::class
    ]);
}
```

In the *CustomerAdminClass* the *New()* method can then be updated to call *getData()* on the *\$customer* object. Thus allowing for the removal of a few lines of code.

```

$form->handleRequest($request);
if($form->isSubmitted() && $form->isValid())
{
    /** @var Customer $customer */
    $customer = $form->getData();

    $customer = new Customer();
    $customer->setFirstName($data['FirstName'])
        ->setLastName($data['LastName'])
        ->setCity($data['City'])
        ->setPhone($data['Phone']);
}

```

When tested, the data is still inserted to the database.

Add Record

First name

Last name

City

Test City

Phone

0192783644

Max 12 Characters! Please remove any spaces.

Add Record!

5	Christina	Berglund	Luleå	0921-12 3465
22	Peter	Davies	SWANSEA	07980002273
23	Testy	McTestFace	Test City	0192783644

This class binding feature of Symfony is worth noting as it allows for custom forms to be created which may wish to insert into many different tables in the database. By simply creating a custom model class with the desired properties and setting the `'data_class'` to the custom model this is made possible.

3.2.4.3 Updating Data

Updating records in the database follows roughly the same pattern as creating records, however, there are some slight differences. Firstly, a new *Edit* method is created in the

CustomerAdminController with a new route specified in the annotations to */edit*, however, when editing data the ID of the record to be edited needs to be passed into the URL to firstly, correctly identify the record to be edited and secondly as this ID will be passed into the function to populate the form's placeholders with the correct data for that record.

```
/**
 * @Route("/edit/{id}")
 */
public function Edit($id, \Symfony\Component\HttpFoundation\Request $request)
{
```

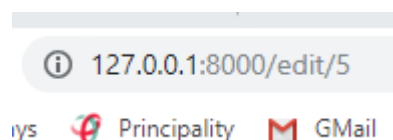
The next steps is to call the *getDoctrine()->getManager()* method with gets Doctrine's entity manager object which will be responsible for retrieving the object from the database. Then, the *EntityManager* is used to call the *getRepository()* method to fetch the correct data from the database relating to the ID which is passed into the function *find()* called on the *getRepository()* method.

```
$em = $this->getDoctrine()->getManager();
$customer = $em->getRepository( className: Customer::class)->find($id);
```

Then, as when creating a form for the insert, a form object is created but for editing data, the fields are populated with the data from the table fetched by the *getRepository()* method in the previous step.

```
$form = $this->createForm( type: CustomerFormType::class, $customer);
```

Now, when the URL */edit/\$id* is visited, where *\$id* represents an ID in the database, the following table is displayed with the data as placeholders in the form to allow users to see the data they are editing.



Add Record

First name

Last name

City

Phone

Max 12 Characters! Please remove any spaces.

The following lines of code in the *Edit()* function are similar to that of the *New()* function to insert data. The *handleRequest()* method handles the POST request and the same *isSubmitted()* function is called in an *if* block to execute the query. The only difference when updating a record is the absence of the *persist()* method. When updating records, only the *flush()* method needs to be called on the entity manager.

```
$form->handleRequest($request);

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

    $em->flush();

    $this->addFlash( type: 'success', message: 'Record Successfully Amended!');

    return $this->redirectToRoute( route: '/customers' );
}

return $this->render( view: 'customerForm', [
    'customerForm' => $form->createView()
]);
```

A new flash message is added to inform the user that the query has executed successfully, and the page is redirected to the customer table view. In the customer view, two new columns have been added to the bootstrap table to render *edit* and *delete* links next to each record. These are passed the corresponding ID using the Twig syntax, meaning that each URL route will have the ID passed into it.

```

<td>{{ customer.Phone }}

```

Surname	City	Phone		
Anders	Berlin	030-0074321	Edit	Delete
Trujillo	México D.F.	(5) 555-4729	Edit	Delete
Moreno	México D.F.	(5) 555-393	Edit	Delete
...	...	...	...	...

Surname	City	Phone		
Anders	Berlin	030-0074321	Edit	Delete
Trujillo	México D.F.	(5) 555-4729	Edit	Delete
Convery	México D.F.	(5) 555-393	Edit	Delete
...	...	...	...	...

Record Successfully Amended!

3.2.4.4 Deleting Data

Deleting records is much simpler than inserting and editing. Again, a method is created, routed to a *del* page which is also passed an ID. The same methods *getDoctrine()*->*getManager()* are called and the data is fetched from the repository with the ID passed in. To remove a record, the method *remove()* is simply called, again the final step is to call the *flush()* method to render the changes to the database.

```

/**
 * @Route("/del/{id}")
 */
public function Delete($id, \Symfony\Component\HttpFoundation\Request $request)
{
    $em = $this->getDoctrine()->getManager();
    $customer = $em->getRepository(Customer::class)->find($id);

    if (!$customer) {
        throw $this->createNotFoundException(
            'No customer found for id '.$id
        );
    }

    $em->remove($customer);
    $em->flush();

    $this->addFlash('success', 'Record Successfully Deleted!');

    return $this->redirectToRoute('app_customeradmin_show');
}

```

PHP Formatters

Record Successfully Deleted!

Customer Information Table

Add Record

Now, when a delete link is clicked in any row, that record is deleted. A new flash message is also added and a redirect.

4. Evaluation

The web application submitted provides a brief look into the capabilities of Symfony using dependency managers, object relational mapping tools, server-side templates and basic CRUD features.

There has also been an attempt to publish the application to Microsoft's Azure portal and although the Git repository is available on Azure DevOps, and a pipeline to Azure hosting services was created, it was not possible to generate the mappings from the application to a MS SQL database in Azure, furthermore, when pushing the application to Azure the CSS files were not included, resulting in a poor looking interface, which can be seen by visiting <https://awdass.azurewebsites.net/>.

The application itself is very simple, and it is disappointing that it does not have any more complex features, such as validation on forms or more than one table interaction, however, this is due to time constraints placed on this assignment with other assignment commitments and the major project deadline also looming.

In hindsight it may not have been the best plan to try and learn a new PHP framework in the short amount of time allocated to complete this assignment, however, the brief encounter with Symfony has proved a good learning exercise and demonstrated briefly how powerful of a framework Symfony is. It is certainly a framework which will be looked at in more detail when time permits.

5. References

- [1] J. Deacon, "Model View Controller (MVC) Architecture," *Comput. Syst. Dev. Consult. Train.*, 2009.
- [2] A. C. Kay, "The Early History of Smalltalk." [Online]. Available: <http://gagne.homedns.org/~tgagne/contrib/EarlyHistoryST.html>. [Accessed: 18-Mar-2019].
- [3] R. Eckstein, "Java SE Application Design With MVC," *Oracle*, 2007. [Online]. Available: <http://www.oracle.com/technetwork/articles/javase/index-142890.html>. [Accessed: 29-Apr-2018].
- [4] T. Reenskaug, "MVC XEROX PARC 1978-79." [Online]. Available: <http://heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html>. [Accessed: 18-Mar-2019].
- [5] S. Burbeck, "Applications Programming in Smalltalk-80 (TM): How to use Model-View-Controller (MVC)," 1987.
- [6] "Model-View-Controller | Microsoft Docs." [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649643\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649643(v=pandp.10)). [Accessed: 18-Mar-2019].
- [7] M. Fowler, *Patterns of Enterprise Application Architecture*. Addison-Wesley, 2003.
- [8] P. Worrall and T. Chaussalet, "Development of a web-based system using the model view controller paradigm to facilitate regional long-term care planning," *Proc. - IEEE Symp. Comput. Med. Syst.*, no. February 2015, 2011.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*. Indianapolis: Addison-Wesley, 1995.
- [10] V. Thakur, .. *ASP.NET 3.5 Application Architecture and Design*. 2008.
- [11] "Page Controller | Microsoft Docs," *Microsoft Developer Network*, 2014. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649595\(v=pandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff649595(v=pandp.10)). [Accessed: 19-Mar-2019].
- [12] "Front Controller | Microsoft Docs," *Microsoft Developer Network*, 2014. [Online]. Available: [https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648617\(v%3Dpandp.10\)](https://docs.microsoft.com/en-us/previous-versions/msp-n-p/ff648617(v%3Dpandp.10)). [Accessed: 19-Mar-2019].
- [13] "Creating and Using Templates (Symfony Docs)," *Symfony Documentation*. [Online]. Available: <https://symfony.com/doc/current/templating.html>. [Accessed: 19-Mar-2019].
- [14] "Razor syntax reference for ASP.NET Core | Microsoft Docs," *Microsoft Developer Network*, 2018. [Online]. Available: <https://docs.microsoft.com/en-us/aspnet/core/mvc/views/razor?view=aspnetcore-2.2>. [Accessed: 19-Mar-2019].

- [15] "What is Symfony." [Online]. Available: <https://symfony.com/what-is-symfony>. [Accessed: 19-Mar-2019].
- [16] "About Symfony Project." [Online]. Available: <https://symfony.com/about>. [Accessed: 20-Mar-2019].
- [17] "Chapter 2 - Exploring Symfony." [Online]. Available: https://symfony.com/legacy/doc/gentle-introduction/1_4/en/02-Exploring-Symfony-s-Code#chapter_02_the_mvc_pattern. [Accessed: 22-Mar-2019].
- [18] M. Noback, *A Year with Symfony*, 1st ed. Leanpub.
- [19] "Understanding how the Front Controller, Kernel and Environments Work together (Symfony Docs)." [Online]. Available: https://symfony.com/doc/current/configuration/front_controllers_and_kernel.html. [Accessed: 22-Mar-2019].
- [20] E. (Betty and) O'neil, *Object/Relational Mapping 2008: Hibernate and the Entity Data Model (EDM)*. .
- [21] "Propel, The Blazing Fast Open-Source PHP 5.5 ORM." [Online]. Available: <http://propelorm.org/>. [Accessed: 22-Mar-2019].
- [22] "Doctrine, The Open-Source PHP ORM and Persistence Tools Project." [Online]. Available: <https://www.doctrine-project.org/>. [Accessed: 22-Mar-2019].
- [23] M. Hoyos, "What is an ORM and Why You Should Use it – Bits and Pieces," *Medium*, 2018. [Online]. Available: <https://blog.bitsrc.io/what-is-an-orm-and-why-you-should-use-it-b2b6f75f5e2a>. [Accessed: 22-Mar-2019].
- [24] M. Sasidharan, "Should I Or Should I Not Use ORM ? – Mithun Sasidharan – Medium," *Medium*, 2016. [Online]. Available: <https://medium.com/@mithunsasidharan/should-i-or-should-i-not-use-orm-4c3742a639ce>. [Accessed: 22-Mar-2019].
- [25] S. Bergmann, "PHPUnit – The PHP Testing Framework." [Online]. Available: <https://phpunit.de/index.html>. [Accessed: 22-Mar-2019].
- [26] "Home - Twig - The flexible, fast, and secure PHP template engine." [Online]. Available: <https://twig.symfony.com/>. [Accessed: 22-Mar-2019].
- [27] N. Latifi, "Architecture patterns in web applications and implementation of find local food," 2010.
- [28] R. T. Fielding and R. N. Taylor, "Principled Design of the Modern Web Architecture."

- [29] A. Rodriguez, "RESTful Web services: The basics," *IBM Developer Works*, 2015. [Online]. Available: <https://www.ibm.com/developerworks/webservices/library/ws-restful/>. [Accessed: 29-Apr-2018].
- [30] P. Japikse, K. Grossnicklaus, and B. Dewey, *Building Web Applications with Visual Studio 2017*, 1st ed. New York: Apress, 2017.
- [31] "Getting Started With FOSRestBundle (Symfony Bundles Docs)." [Online]. Available: <https://symfony.com/doc/master/bundles/FOSRestBundle/index.html>. [Accessed: 22-Mar-2019].
- [32] "Symfony 4: Let's Launch! Stellar Development with Symfony 4 | SymfonyCasts," *Symfony Casts*, 2017. [Online]. Available: <https://symfonycasts.com/screencast/symfony/setup#play>. [Accessed: 20-Mar-2019].
- [33] S. Armand, *Extending Symfony2 Web Application Framework*. Birmingham, Mumbai: Packt Publishing, 2014.
- [34] "Git - About Version Control," *Git SCM*. [Online]. Available: <https://git-scm.com/book/en/v1/Getting-Started-About-Version-Control>. [Accessed: 18-Jan-2019].
- [35] P. Davies, "Projects - Home," *Azure DevOps*, 2019. [Online]. Available: <https://dev.azure.com/pdavies079868/>. [Accessed: 20-Mar-2019].
- [36] GitKraken, "Why Users Love GitKraken - Customer Stories | GitKraken." [Online]. Available: <https://www.gitkraken.com/customers>. [Accessed: 13-Feb-2019].
- [37] GitHub, "GitHub Student Developer Pack." [Online]. Available: <https://education.github.com/students>. [Accessed: 13-Feb-2019].
- [38] "Azure pipelines | Microsoft Azure," *Azure DevOps Pipelines*. [Online]. Available: <https://azure.microsoft.com/en-gb/services/devops/pipelines/>. [Accessed: 20-Mar-2019].
- [39] "PhpStorm: The Lightning-Smart IDE for PHP Programming by JetBrains." [Online]. Available: <https://www.jetbrains.com/phpstorm/>. [Accessed: 22-Mar-2019].
- [40] "SQL Server Example Database." [Online]. Available: <https://www.dofactory.com/sql/sample-database>. [Accessed: 22-Mar-2019].
- [41] "Bare - Bootstrap Starter Template - Start Bootstrap." [Online]. Available: <https://startbootstrap.com/templates/bare/>. [Accessed: 25-Mar-2019].
- [42] "Tryit Editor v3.6 - Bootstrap Table," *W3 Schools*. [Online]. Available: https://www.w3schools.com/bootstrap/tryit.asp?filename=trybs_ref_table-responsive&stacked=h. [Accessed: 25-Mar-2019].

6. Bibliography

"Home - Twig - The flexible, fast, and secure PHP template engine." [Online]. Available: <https://twig.symfony.com/>. [Accessed: 22-Mar-2019].

"Databases and the Doctrine ORM (Symfony Docs)." [Online]. Available: <https://symfony.com/doc/current/doctrine.html>. [Accessed: 26-Mar-2019].

"FormType Field (Symfony Docs)." [Online]. Available: <https://symfony.com/doc/current/reference/forms/types/form.html>. [Accessed: 26-Mar-2019].

"How to Customize Form Rendering (Symfony Docs)." [Online]. Available: https://symfony.com/doc/current/form/form_customization.html#form-end-form-view. [Accessed: 26-Mar-2019].

"Success (Flash) Messages & Symfony 4 Forms: Build, Render & Conquer! | SymfonyCasts." [Online]. Available: <https://symfonycasts.com/screencast/symfony-forms/flash-messages#play>. [Accessed: 26-Mar-2019].

"Updating Existing Doctrine Entities." [Online]. Available: <https://codereviewvideos.com/course/beginner-s-guide-to-symfony-3-forms/video/updating-existing-doctrine-entities>. [Accessed: 26-Mar-2019].

"How to Create REST API in Symfony 3.1," *Cloud Ways*, 2016. [Online]. Available: <https://www.cloudways.com/blog/rest-api-in-symfony-3-1/>. [Accessed: 22-Mar-2019].

"Entity Framework | Microsoft Docs." [Online]. Available: <https://docs.microsoft.com/en-us/ef/>. [Accessed: 22-Mar-2019].

"Chapter 8 - Inside The Model Layer (Doctrine) (1_4)." [Online]. Available: https://symfony.com/legacy/doc/gentle-introduction/1_4/en/08-Inside-the-Model-Layer-Doctrine. [Accessed: 22-Mar-2019].

"Deploy PHP application to Azure App Service using Azure Pipelines | Azure DevOps Hands-on-Labs." [Online]. Available: <https://www.azuredevopslabs.com/labs/vstsextend/PHP/>. [Accessed: 20-Mar-2019].

"Introduction - Composer," *Composer*. [Online]. Available: <https://getcomposer.org/doc/00-intro.md>. [Accessed: 20-Mar-2019].

G. Sironi, "Practical PHP Patterns: Page Controller - DZone Web Dev," *DZone*. [Online]. Available: <https://dzone.com/articles/practical-php-patterns/practical-php-patterns-page>. [Accessed: 19-Mar-2019].

D. Upton, *Codeigniter for Rapid Php Application Development*. 2007.

C. Pitt, *Pro PHP MVC*. Apress, 2012.

- P. R. Pilaka, "Aspiring Architect: Page Controller Vs Front Controller Pattern," 2011. [Online]. Available: <http://pratapreddypilaka.blogspot.com/2011/09/page-controller-vs-front-controller.html>. [Accessed: 19-Mar-2019].
- SesnioLabs, "The Book Version: 2.8," 2016.
- C.-H. Chang, C.-W. Lu, W. C. Chu, N.-L. Hsueh, and C.-S. Koong, "A Case Study of Pattern-based Software Framework to Improve the Quality of Software Development."
- A. Freeman, *Pro ASP. NET Core MVC*. 2016.
- J. Schekkerman, *How to Survive in the Jungle of Enterprise Architecture Frameworks: Creating or Choosing an Enterprise Architecture Framework*, 2nd ed. Trafford Publishing, 2006.
- D. Greefhorst and E. Proper, *Architecture Principles: The Cornerstones of Enterprise Architecture*, 1st ed. Springer, 2011.
- P. Raj, A. Raman, and H. Subramanian, *Architectural Patterns Uncover essential patterns in the most indispensable realm of enterprise architecture*. 2017.
- J. D. Herrington, "Five common PHP design patterns," 2006.

7. Appendix

Base Template

```
<!DOCTYPE html>
<html>
  <head>
    {% block stylesheets %}
      <link href="{{ asset('assets/css/bootstrap.css') }}"
rel="stylesheet"/>
      <link href="{{ asset('assets/css/styles.css') }}"
rel="stylesheet"/>
    {% endblock %}
    <meta charset="UTF-8">
    <title>{% block title %}Welcome to AWD{% endblock %}</title>
  </head>
  <header>
    <!-- Main Header -->
    <nav class="navbar navbar-expand-lg navbar-dark bg-dark static-top">
      <div class="container">
        <a class="navbar-brand" href="#">AWD Administration</a>
        <button class="navbar-toggler" type="button" data-
toggle="collapse" data-target="#navbarResponsive" aria-
controls="navbarResponsive" aria-expanded="false" aria-label="Toggle
navigation">

          <span class="navbar-toggler-icon"></span>
        </button>
        <div class="collapse navbar-collapse" id="navbarResponsive">
          <ul class="navbar-nav ml-auto">
            <li class="nav-item active">
              <a class="nav-link" href="/">Home
                <span class="sr-only">(current)</span>
              </a>
            </li>
          </ul>
        </div>
      </div>
    </nav>
  </header>
  <body>
    {% for message in app.flashes('success') %}
      <div class="container">
        <div class="alert alert-success">
          {{ message }}
        </div>
      </div>
    {% endfor %}
    {% block body %}{% endblock %}
    {% block form %}{% endblock %}
  </body>
  <footer></footer>
</html>
```

Home Template

```
{% extends 'base.html.twig' %}

{% block body %}
    <!-- Page Content -->
    <div class="container">

        <!-- Page Heading -->
        <h1 class="my-4">Welcome -
            <small>AWD Administration Panel</small>
        </h1>

        <div class="row">
            <div class="col-lg-4 col-sm-6 mb-4">
                <div class="card h-100">
                    <a href="#"></a>
                    <div class="card-body">
                        <h4 class="card-title">
                            <a href="/customers">Customers</a>
                        </h4>
                        <br>
                        <p class="card-text"><b>Customer
Information:</b><br><br>Names, City's & Phone Numbers</p>
                    </div>
                </div>
            </div>
            <div class="col-lg-4 col-sm-6 mb-4">
                <div class="card h-100">
                    <a href="#"></a>
                    <div class="card-body">
                        <h4 class="card-title">
                            <a href="">Orders</a>
                        </h4>
                        <br>
                        <p class="card-text"><b>Order
Information:</b><br><br>Order Dates, Order Numbers, Customer Id's & Order
Totals</p>
                    </div>
                </div>
            </div>
            <div class="col-lg-4 col-sm-6 mb-4">
                <div class="card h-100">
                    <a href="#"></a>
                    <div class="card-body">
                        <h4 class="card-title">
                            <a href="#">Order Items</a>
                        </h4>
                        <br>
                        <p class="card-text"><b>Items Ordered by the
Customer:</b><br><br>Order Id's, Product Id's, Unit Prices & Quantity's
Ordered</p>
                    </div>
                </div>
            </div>
            <div class="col-lg-4 col-sm-6 mb-4">
                <div class="card h-100">
                    <a href="#"></a>
                    <div class="card-body">
                        <h4 class="card-title">
                            <a href="#">Products</a>
                        </h4>
                    </div>
                </div>
            </div>
        </div>
    </div>
{% endblock %}
```

```

        <br>
        <p class="card-text"><b>Products in
Stock:</b><br><br>Product Name's, Suppliers, Unit Prices, Packaging
Information & Continuation</p>
    </div>
</div>
</div>
<div class="col-lg-4 col-sm-6 mb-4">
    <div class="card h-100">
        <a href="#"></a>
        <div class="card-body">
            <h4 class="card-title">
                <a href="#">Suppliers</a>
            </h4>
            <br>
            <p class="card-text"><b>Supplier
Information:</b><br><br>Company Names, Contact Names, Contact Titles,
Cities, Country's, Phone & Fax Numbers</p>
        </div>
    </div>
</div>
</div>
<!-- /.row -->

</div>
<!-- /.container -->

{% endblock %}

```

Customers Template

```
{% extends 'base.html.twig' %}

{% block body %}
<h1></h1>

<div class="container">

</div>
    <!-- Table -->
    <div class="container">
        <br><br>
        <h2>Customer Information Table</h2>
        <a class="button-link" href="/addCustomer">Add Record</a>
        <br><br>
        <div class="table-responsive">
            <table class="table">
                <thead>
                    <tr>
                        <th>ID</th>
                        <th>First Name</th>
                        <th>Surname</th>
                        <th>City</th>
                        <th>Phone</th>
                    </tr>
                </thead>
                <tbody>
                    {% for customer in Customers %}
                    <tr>
                        <td>{{ customer.Id }}</td>
                        <td>{{ customer.firstName }}</td>
                        <td>{{ customer.lastName }}</td>
                        <td>{{ customer.City }}</td>
                        <td>{{ customer.Phone }}</td>
                        <td><a href="/edit/{{ customer.Id }}">Edit</a></td>
                        <td><a href="/del/{{ customer.Id }}">Delete</a></td>
                    </tr>
                    {% endfor %}
                </tbody>
            </table>
        </div>
    </div>
    <!-- /Table -->

{% endblock %}
```

Customer Form Template

```
{% extends 'base.html.twig' %}

{% block form %}

    <!-- Form -->
    <div class="container">

        <a class="button-link record-button" href="/customers"><
Back</a>

        <div class="text-center border border-light p-5">
            <p class="h4 mb-4">Add Record</p>
            {{ form_start(customerForm) }}
            <div class="form-row mb-4">
                <div class="col">
                    <!-- First name -->
                    {{ form_row(customerForm.FirstName) }}
                </div>
                <div class="col">
                    <!-- Last name -->
                    {{ form_row(customerForm.LastName) }}
                </div>
            </div>
            {{ form_row(customerForm.City) }}
            <small id="spacer" class="form-text text-muted mb-
4"></small>

                {{ form_row(customerForm.Phone) }}
            <small id="guidance" class="form-text text-muted mb-4">
                Max 12 Characters! Please remove any spaces.
            </small>
            {{ form_widget(customerForm) }}
            <button class="btn btn-info my-4 btn-block"
type="submit">Add Record</button>
            {{ form_end(customerForm) }}
        </div>
    <!-- /Form -->
{% endblock %}
```

Customer Repository

```
<?php

namespace App\Repository;

use App\Entity\Customer;
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Symfony\Bridge\Doctrine\RegistryInterface;

/**
 * @method Customer|null find($id, $lockMode = null, $lockVersion = null)
 * @method Customer|null findOneBy(array $criteria, array $orderBy = null)
 * @method Customer[]    findAll()
 * @method Customer[]    findBy(array $criteria, array $orderBy = null,
$limit = null, $offset = null)
 */
class CustomerRepository extends ServiceEntityRepository
{
    public function __construct(RegistryInterface $registry)
    {
        parent::__construct($registry, Customer::class);
    }

    /**
     * @return Customer[] Returns an array of Customer objects
     */
    public function findByExampleField($value)
    {
        return $this->createQueryBuilder('c')
            ->andWhere('c.exampleField = :val')
            ->setParameter('val', $value)
            ->orderBy('c.id', 'ASC')
            ->setMaxResults(10)
            ->getQuery()
            ->getResult();
    }

    /**
     *
     */
    public function findOneBySomeField($value): ?Customer
    {
        return $this->createQueryBuilder('c')
            ->andWhere('c.exampleField = :val')
            ->setParameter('val', $value)
            ->getQuery()
            ->getOneOrNullResult();
    }
}
```

Customer Form Type Class

```
<?php
/**
 * Created by PhpStorm.
 * User: Peter
 * Date: 25/03/2019
 * Time: 20:11
 */

namespace App\Form;

use App\Entity\Customer;
use function Sodium\add;
use Symfony\Component\Form\AbstractType;
use Symfony\Component\Form\FormBuilderInterface;
use Symfony\Component\Form\Extension\Core\Type\FormType;
use Symfony\Component\Form\Extension\Core\Type\TextType;
use Symfony\Component\OptionsResolver\OptionsResolver;

class CustomerFormType extends AbstractType
{
    public function buildForm(FormBuilderInterface $builder, array $options)
    {
        $builder
            ->add('FirstName', TextType::class, array('attr' =>
array('class' => 'form-control')))
            ->add('LastName', TextType::class, array('attr' => array('class'
=> 'form-control')))
            ->add('City', TextType::class, array('attr' => array('class' =>
'form-control')))
            ->add('Phone', TextType::class, array('attr' => array('class' =>
'form-control')));
    }

    public function configureOptions(OptionsResolver $resolver)
    {
        $resolver->setDefaults([

            'data_class' => Customer::class
        ]);
    }
}
```

Customer Entity Model Class

```
<?php

namespace App\Entity;

use Doctrine\ORM\Mapping as ORM;

/**
 * @ORM\Entity(repositoryClass="App\Repository\CustomerRepository")
 */
class Customer
{
    /**
     * @ORM\Id()
     * @ORM\GeneratedValue()
     * @ORM\Column(type="integer")
     */
    private $id;

    /**
     * @ORM\Column(type="string", length=40)
     */
    private $firstName;

    /**
     * @ORM\Column(type="string", length=40)
     */
    private $lastName;

    /**
     * @ORM\Column(type="string", length=100, nullable=true)
     */
    private $city;

    /**
     * @ORM\Column(type="string", length=12, nullable=true)
     */
    private $phone;

    public function getId(): ?int
    {
        return $this->id;
    }

    public function getFirstName(): ?string
    {
        return $this->firstName;
    }

    public function setFirstName(string $firstName): self
    {
        $this->firstName = $firstName;

        return $this;
    }

    public function getLastName(): ?string
    {
        return $this->lastName;
    }

    public function setLastName(string $lastName): self
    {
        $this->lastName = $lastName;
    }
}
```

```
        return $this;
    }

    public function getCity(): ?string
    {
        return $this->City;
    }

    public function setCity(?string $City): self
    {
        $this->City = $City;

        return $this;
    }

    public function getPhone(): ?string
    {
        return $this->Phone;
    }

    public function setPhone(?string $Phone): self
    {
        $this->Phone = $Phone;

        return $this;
    }
}
```

Home Page Controller

```
<?php
/**
 * Created by PhpStorm.
 * User: Pet
 * Date: 2019-03-21
 * Time: 13:22
 */

namespace App\Controller;

use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;

class HomeController extends AbstractController // Need AbstractController
to render template
{
    // Route Annotations
    /**
     * @Route("/")
     */
    public function HomePage()
    {
        return $this->render('home.html.twig'); // Return a HTML view
    }

    /**
     * @Route("/home")
     */
    public function Show()
    {
        return new Response('Home');
    }

    /**
     * @Route("/home1")
     */
    public function ShowTwig()
    {
        return $this->render('customers.html.twig'); // Return a HTML view
    }

    /**
     * @Route("/homes/{slug}")
     */
    public function Shows($slug)
    {
        return new Response(sprintf('Future page to show: %s', $slug));
    }

    // Controllers must return response object.
    // Make database queiries or render template or call API's
}
```

Customer Admin Page Controller

```
<?php
/**
 * Created by PhpStorm.
 * User: Peter
 * Date: 25/03/2019
 * Time: 17:05
 */

namespace App\Controller;

use App\Form\CustomerFormType;
use Doctrine\ORM\EntityManagerInterface;
use http\Env\Request;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use App\Entity\Customer;
use Symfony\Component\HttpFoundation\RedirectResponse;

class CustomerAdminController extends AbstractController
{
    /**
     * @Route("/addCustomer")
     */
    public function New(EntityManagerInterface $em,
        \Symfony\Component\HttpFoundation\Request $request) // Only Works like this,
        with full path??
    {
        $form = $this->createForm(CustomerFormType::class);

        $form->handleRequest($request);
        if($form->isSubmitted() && $form->isValid())
        {
            /** @var Customer $customer */
            $customer = $form->getData();

            // $customer = new Customer();
            // $customer->setFirstName($data['FirstName'])
            // $customer->setLastName($data['LastName'])
            // $customer->setCity($data['City'])
            // $customer->setPhone($data['Phone']);

            $em->persist($customer);
            $em->flush();

            $this->addFlash('success', 'Record Successfully Added!');

            return $this->redirectToRoute('app_customeradmin_show');
        }

        return $this->render('customerForm.html.twig', [
            'customerForm' => $form->createView()
        ]);
    }

    /**
     * @Route("/customers")
     */
    public function Show(EntityManagerInterface $em)
    {
        $repo = $em->getRepository(Customer::class);
        /** @var Customer $customer */
    }
}
```

```

        $customer = $repo->findAll();

        // If nothing in database throw exception
        if(!$customer)
        {
            throw $this->createNotFoundException(sprintf('No records
found!'));
        }

        return $this->render('customers.html.twig', [

            'Customers' => $customer

        ]);
    }

    /**
     * @Route("/edit/{id}")
     */
    public function Edit($id, \Symfony\Component\HttpFoundation\Request
$request)
    {
        $em = $this->getDoctrine()->getManager();
        $customer = $em->getRepository(Customer::class)->find($id);

        $form = $this->createForm(CustomerFormType::class, $customer);

        $form->handleRequest($request);

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

            $em->flush();

            $this->addFlash('success', 'Record Successfully Amended!');

            return $this->redirectToRoute('app_customeradmin_show');

        }

        return $this->render('customerForm.html.twig', [
            'customerForm' => $form->createView()
        ]);
    }

    /**
     * @Route("/del/{id}")
     */
    public function Delete($id, \Symfony\Component\HttpFoundation\Request
$request)
    {
        $em = $this->getDoctrine()->getManager();
        $customer = $em->getRepository(Customer::class)->find($id);

        if (!$customer) {
            throw $this->createNotFoundException(
                'No customer found for id '.$id
            );
        }

        $em->remove($customer);
        $em->flush();

        $this->addFlash('success', 'Record Successfully Deleted!');

        return $this->redirectToRoute('app_customeradmin_show');
    }

```

```
}  
}
```

Index.php Front Controller

```
<?php

use App\Kernel;
use Symfony\Component\Debug\Debug;
use Symfony\Component\HttpFoundation\Request;

require dirname(__DIR__).'/config/bootstrap.php';

if ($_SERVER['APP_DEBUG']) {
    umask(0000);

    Debug::enable();
}

if ($trustedProxies = $_SERVER['TRUSTED_PROXIES'] ??
    $_ENV['TRUSTED_PROXIES'] ?? false) {
    Request::setTrustedProxies(explode(',', $trustedProxies),
        Request::HEADER_X_FORWARDED_ALL ^ Request::HEADER_X_FORWARDED_HOST);
}

if ($trustedHosts = $_SERVER['TRUSTED_HOSTS'] ?? $_ENV['TRUSTED_HOSTS'] ??
    false) {
    Request::setTrustedHosts([$trustedHosts]);
}

$kernel = new Kernel($_SERVER['APP_ENV'], (bool) $_SERVER['APP_DEBUG']);
$request = Request::createFromGlobals();
$response = $kernel->handle($request);
$response->send();
$kernel->terminate($request, $response);
```

.ENV Config File

```
# In all environments, the following files are loaded if they exist,
# the later taking precedence over the former:
#
# * .env                contains default values for the environment
variables needed by the app
# * .env.local          uncommitted file with local overrides
# * .env.$APP_ENV       committed environment-specific defaults
# * .env.$APP_ENV.local uncommitted environment-specific overrides
#
# Real environment variables win over .env files.
#
# DO NOT DEFINE PRODUCTION SECRETS IN THIS FILE NOR IN ANY OTHER COMMITTED
FILES.
#
# Run "composer dump-env prod" to compile .env files for production use
(requires symfony/flex >=1.2).
#
https://symfony.com/doc/current/best\_practices/configuration.html#infrastructure-related-configuration

###> symfony/framework-bundle ###
APP_ENV=dev
APP_SECRET=245ea120f3704a9266278c483244171c
#TRUSTED_PROXIES=127.0.0.1,127.0.0.2
#TRUSTED_HOSTS='^localhost|example\.com$'
###< symfony/framework-bundle ###

###> doctrine/doctrine-bundle ###
# Format described at http://docs.doctrine-project.org/projects/doctrine-dbal/en/latest/reference/configuration.html#connecting-using-a-url
# For an SQLite database, use: "sqlite:///kernel.project_dir%/var/data.db"
# Configure your db driver and server_version in
config/packages/doctrine.yaml
DATABASE_URL=mysql://root:p079868@127.0.0.1:3306/awd_shop
###< doctrine/doctrine-bundle ###
```