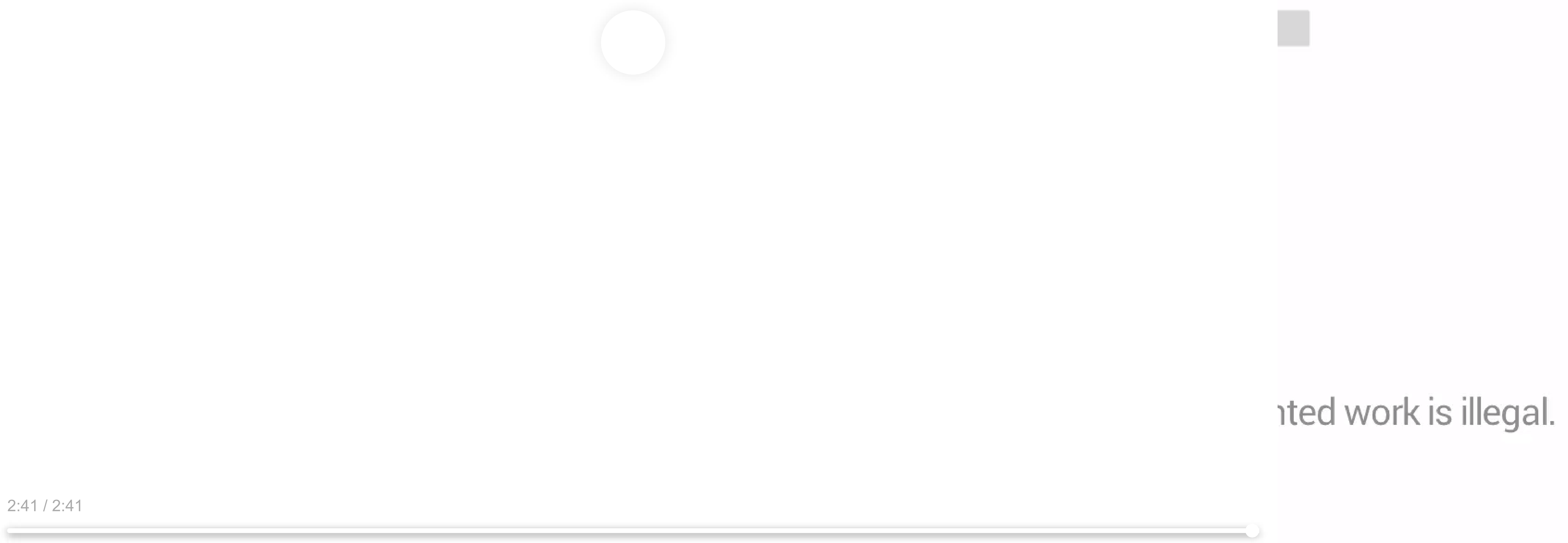
Symfony | Day 2 | Pre-Work

Symfony
DAY 2

Table of contents:

Introduction



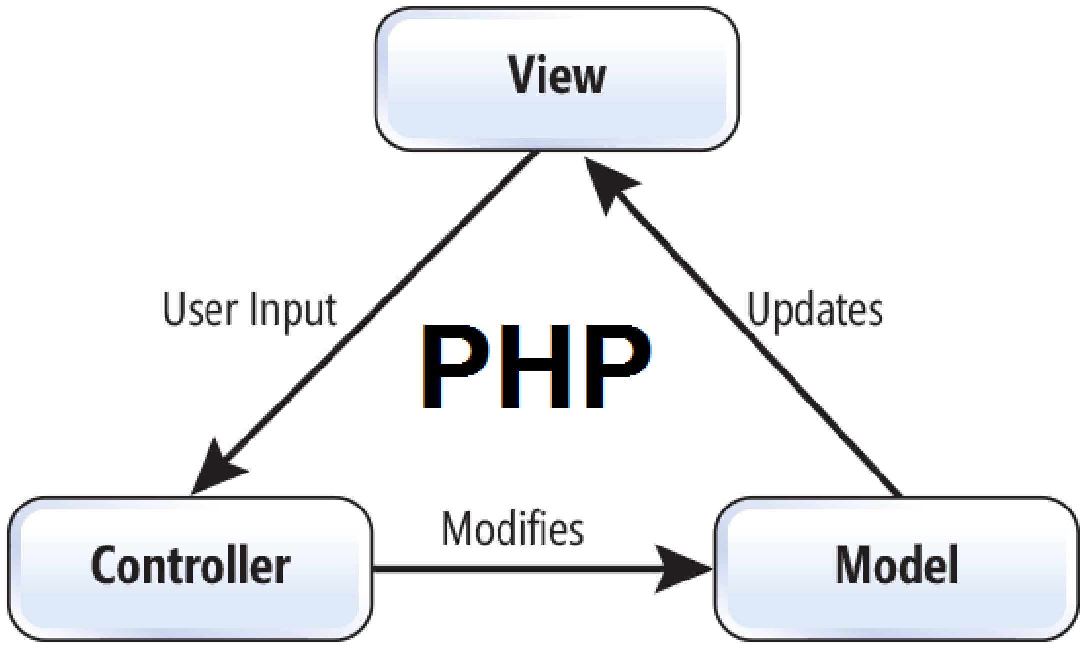
At first glance, the code behind a symfony-driven application can seem quite daunting. It consists of many directories and scripts, and the files are a mix of PHP classes, HTML, and even an intermingling of the two. You'll also see references to classes that are otherwise nowhere to be found within the application folder, and the directory depth stretches to six levels. But once you understand the reason behind all of this seeming complexity, you'll suddenly feel like it's so natural that you wouldn't trade the symfony application structure for any other.

The MVC pattern

As you already know, Symfony is based on the classic web design pattern known as the MVC architecture, which consists of three levels:

- **The Model** represents the information on which the application operates-its business logic
- **The View** renders the model into a web page suitable for interaction with the user
- **The Controller** responds to user actions and invokes changes on the model or view as appropriate

This image illustrates the MVC pattern:



The MVC architecture separates the business logic (model) and the presentation (view), resulting in greater maintainability. For instance, if your application should run on both standard web browsers and handheld devices, you just need a new view; you can keep the original controller and model. The controller helps to hide the detail of the protocol used for the request (HTTP, console mode, mail, and so on) from the model and the view. And the model abstracts the logic of the data, which makes the view and the action independent of, for instance, the type of database used by the application.

MVC Layering

To help you understand MVC advantages, let's see how to convert a basic PHP application to an MVC-architectured application. A list of posts for a weblog application will be a perfect example.

Flat Programming

In a flat PHP file, displaying a list of database entries might look like the script presented in the example below:

First create a database and give it a name (dbtest).

Then create a table named post:

```
CREATE TABLE post(post_id int(11) AUTO_INCREMENT PRIMARY
KEY,
               post_title varchar(20) not null,
               post_date date)
```

```
<?php

// Connecting, selecting database
$link= mysqli_connect('localhost', 'root', '', 'dbtest');
// Performing SQL query
$sql="SELECT * FROM post";
$result = mysqli_query($link, $sql);
?>

<html>
 <head>
   <title>List of Posts</title>
 </head>
 <body>
  <h1>List of Posts</h1>
  <table>
    <tr><th>Date</th><th>Title</th></tr>
<?php
// Printing results in HTML
while ($row = mysqli_fetch_array($result))
{
echo "\t<tr>\n";
printf("\t\t<td> %s </td>\n", $row['post_date']);
printf("\t\t<td> %s </td>\n", $row['post_title']);
echo "\t</tr>\n";
}
?>
   </table>
 </body>
</html>

<?php
// Closing connection
mysqli_close($link);
?>
```

That's quick to write, fast to execute, and impossible to maintain. The major problems with this code:

- There is no error-checking (what if the connection to the database fails?)
- HTML and PHP code are mixed, even interwoven together
- The code is tied to a MySQL database

Isolating the Presentation

The echo and printf calls in the example above make the code difficult to read. Modifying the HTML code to enhance the presentation is a hassle with the current syntax. So the code can be split into two parts. First, the pure PHP code with all the business logic goes in a controller script, as shown in the example below:

controller.php

```php
<?php

// Connecting, selecting database
$link= mysqli_connect('localhost', 'root', '', 'dbtest');
// Performing SQL query
$result = mysqli_query($link, 'SELECT * FROM post');

// Filling up the array for the view
$posts = array();
while ($row = mysqli_fetch_array($result))
{
 $posts[] = $row;
}

// Closing connection
mysqli_close($link);

// Requiring the view
require('view.php');
```

The HTML code, containing template-like PHP syntax, is stored in a view script, as shown in the example below:

view.php :

```
<!DOCTYPE html>
<html>
 <head>
   <title>List of Posts</title>
 </head>
 <body>
   <h1>List of Posts</h1>
   <table>
     <tr><th>Date</th><th>Title</th></tr>
   <?php foreach ($posts as $post): ?>
     <tr>
       <td><?php echo $post['post_date'] ?></td>
       <td><?php echo $post['post_title'] ?></td>
     </tr>
   <?php endforeach; ?>
   </table>
 </body>
</html>
```

A good rule of thumb to determine whether the view is clean enough is that it should contain only a minimum amount of PHP code, in order to be understood by an HTML designer without PHP knowledge. The most common statements in views are echo, if/endif, foreach/endforeach, and that's about all. Also, there should not be PHP code echoing HTML tags.
All the logic is moved to the controller script, and contains only pure PHP code, with no HTML inside. As a matter of fact, you should imagine that the same controller could be reused for a totally different presentation, perhaps in a PDF file or an XML structure.

Isolating the Data Manipulation

Most of the controller script code is dedicated to data manipulation. But what if you need the list of posts for another controller, say one that would output an RSS feed of the weblog posts? What if you want to keep all the database queries in one place, to avoid code duplication? What if you decide to change the data model so that the post table gets renamed weblog_post? What if you want to switch to PostgreSQL instead of MySQL? In order to make all that possible, you need to remove the data-manipulation code from the controller and put it in another script, called the model, as shown in the example below:

```php
<?php
function getAllPosts()
{
 // Connecting, selecting database
 $link= mysqli_connect('localhost', 'root', '','dbtest');

 // Performing SQL query
 $query='SELECT * FROM post';
 $result = mysqli_query($link, $query);

 // Filling up the array
 $posts = array();
 while ($row = mysqli_fetch_array($result))
 {
  $posts[] = $row;
 }

 // Closing connection
 mysqli_close($link);
 return $posts;
}
```

The revised controller is presented in the example below:

```php
<?php

// Requiring the model
require_once('model.php');

// Retrieving the list of posts
$posts = getAllPosts();

// Requiring the view
require_once('view.php');
```

The controller becomes easier to read. Its sole task is to get the data from the model and pass it to the view. In more complex applications, the controller also deals with the request, the user session, the authentication, and so on. The use of explicit names for the functions of the model even makes code comments unnecessary in the controller.
The model script is dedicated to data access and can be organized accordingly. All parameters that don't depend on the data layer (like request parameters) must be given by the controller and not accessed directly by the model. The model functions can be easily reused in another controller.

Layer Separation Beyond MVC

So the principle of the MVC architecture is to separate the code into three layers, according to its nature. Data logic code is placed within the model, presentation code within the view, and application logic within the controller.
Other additional design patterns can make the coding experience even easier. The model, view, and controller layers can be further subdivided.

Database Abstraction

The model layer can be split into a data access layer and a database abstraction layer. That way, data access functions will not use database-dependent query statements, but call some other functions that will do the queries themselves. If you change your database system later, only the database abstraction layer will need updating..
A sample database abstraction layer is presented in the example below

Model.php :

```php
<?php
function open_connection($host, $user, $password)
{
return mysqli_connect($host, $user, $password);
}
function close_connection($link)
{
mysqli_close($link);
}
function query_database($query, $database, $link)
{
mysqli_select_db($link, $database);
return mysqli_query($link, $query);
}
function fetch_results($result)
{
return mysqli_fetch_array($result);
}
```

An example of a MySQL-specific data access layer:

```
function getAllPosts()
{
// Connecting to database
$link= open_connection('localhost', 'root', '');
// Performing SQL query
$result = query_database('SELECT post_date, post_title FROM
post', 'dbtest', $link);
// Filling up the array
$posts = array();
while ($row = fetch_results($result))
{
   $posts[] = $row;
}
// Closing connection
close_connection($link);
return $posts;
}
```

You can check that no database-engine dependent functions can be found in the data access layer, making it database-independent. Additionally, the functions created in the database abstraction layer can be reused for many other model functions that need access to the database.
The examples above are still not very satisfactory, and there is some work left to do to have a full database abstraction (abstracting the SQL code through a database-independent query builder, moving all functions into a class, and so on). But the purpose of these examples is not to show you how to write all that code by hand, and you will see that symfony natively does all the abstraction very well.

View Elements

The view layer can also benefit from some code separation. A web page often contains consistent elements throughout an application: the page headers, the graphical layout, the footer, and the global navigation. Only the inner part of the page changes. That's why the view is separated into a layout and a template. The layout is usually global to the application, or to a group of pages. The template only puts in shape the variables made available by the controller. Some logic is needed to make these components work together, and this view logic layer will keep the name view. According to these principles, the view part of example above can be separated into three parts, as shown in the three following examples:

The template part of the view:

```
<h1>List of Posts</h1>
<table>
<tr><th>Date</th><th>Title</th></tr>
<?php foreach ($posts as $post): ?>
 <tr>
   <td><?php echo $post['post_date'] ?></td>
   <td><?php echo $post['post_title'] ?></td>
 </tr>
<?php endforeach; ?>
</table>
```

The view logic part of the view:

```php
<?php

$title = 'List of Posts';
$posts = getAllPosts();
```

The layout part of the view:

```html
<html>
<head>
  <title><?php echo $title ?></title>
</head>
<body>
  <?php include('mytemplate.php'); ?>
</body>
</html>
```

Action and front controller

The controller doesn't do much in the previous example, but in real web applications, the controller has a lot of work. An important part of this work is common to all the controllers of the application. The common tasks include request handling, security handling, loading the application configuration, and similar chores. This is why the controller is often divided into a front controller, which is unique for the whole application, and actions, which contain only the controller code specific to one page.
One of the great advantages of a front controller is that it offers a unique entry point to the whole application. If you ever decide to close the access to the application, you will just need to edit the front controller script. In an application without a front controller, each individual controller would need to be turned off.

Object Orientation

All the previous examples use procedural programming. As you already know the OOP capabilities of modern languages make the programming even easier, since objects can encapsulate logic, inherit from one another, and provide clean naming conventions.
Implementing an MVC architecture in a language that is not object-oriented raises namespace and code-duplication issues, and the overall code is difficult to read.
Object orientation allows developers to deal with such things as the view object, the controller object, and the model classes, and to transform all the functions in the previous examples into methods. It is a must for MVC architectures.

Last modified: Wednesday, 11 July 2018, 9:56 AM