MVC And Frameworks in a PHP Web Application

Internet Applications, ID1354

Contents

- MVC in a PHP Web Application
- The id1354-fw Framework

Section

- MVC in a PHP Web Application
- The id1354-fw Framework

Object Oriented Design!

- We want the code to be easy to modify and easy to understand. To achieve this we need (among other things):

Object Oriented Design!

- We want the code to be easy to modify and easy to understand. To achieve this we need (among other things):
- High Cohesion, Each class, method, etc has well-defined knowledge and a well-defined task.

Object Oriented Design!

- We want the code to be easy to modify and easy to understand. To achieve this we need (among other things):
- High Cohesion, Each class, method, etc has well-defined knowledge and a well-defined task.
- Low coupling, Objects and subsystems do not depend on each other more than necessary.

Object Oriented Design!

- We want the code to be easy to modify and easy to understand. To achieve this we need (among other things):
- High Cohesion, Each class, method, etc has well-defined knowledge and a well-defined task.
- Low coupling, Objects and subsystems do not depend on each other more than necessary.
- Encapsulation, Objects and subsystems do not reveal their internals.

The MVC Architectural Pattern

- The MVC pattern states that the application contains the layers Model, View and Controller.

The MVC Architectural Pattern

- The MVC pattern states that the application contains the layers Model, View and Controller.
- View contains all code related to the user interface, but no other code. User interface code includes both code that generates a UI and code that interprets user actions.

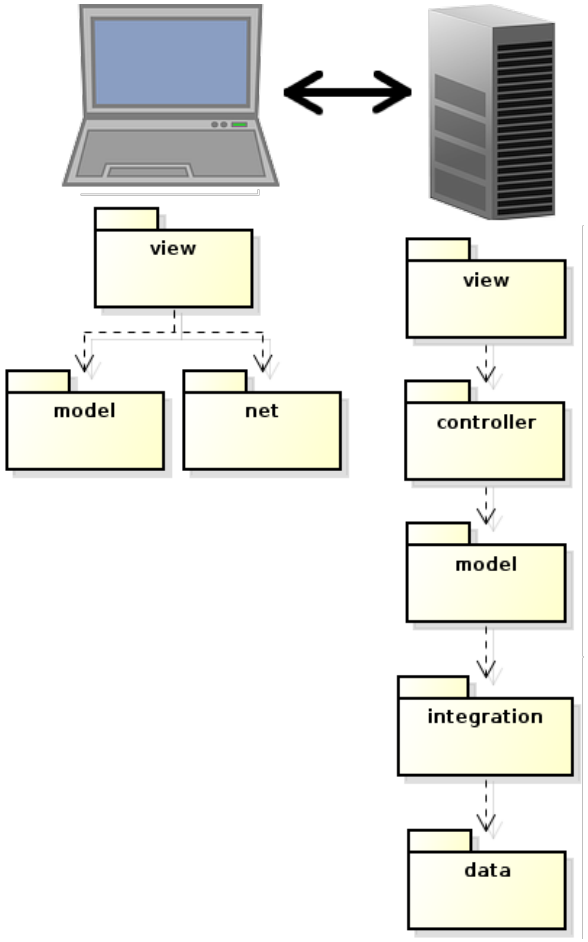
The MVC Architectural Pattern

- The MVC pattern states that the application contains the layers Model, View and Controller.
- View contains all code related to the user interface, but no other code. User interface code includes both code that generates a UI and code that interprets user actions.
- Model contains all data and methods that operate on the data. This is the actual functionality of the application.

The MVC Architectural Pattern

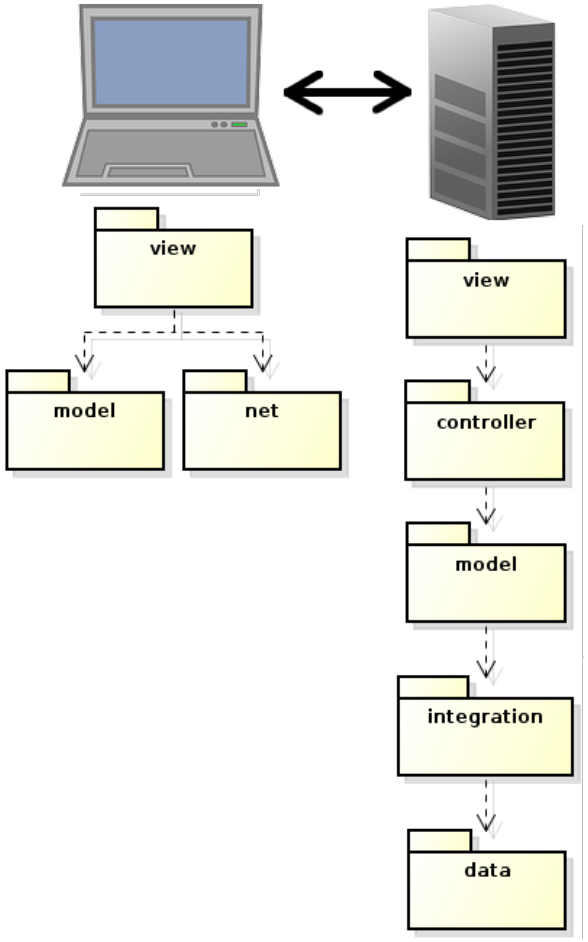
- The MVC pattern states that the application contains the layers Model, View and Controller.
- View contains all code related to the user interface, but no other code. User interface code includes both code that generates a UI and code that interprets user actions.
- Model contains all data and methods that operate on the data. This is the actual functionality of the application.
- Controller is an intermediary between View and Model. Each user action should correspond to one method call from view to controller. It is the task of the controller to know in detail which objects and methods in the model should be called (and in which order) to perform a particular task.

Remember: Server-Side Layers



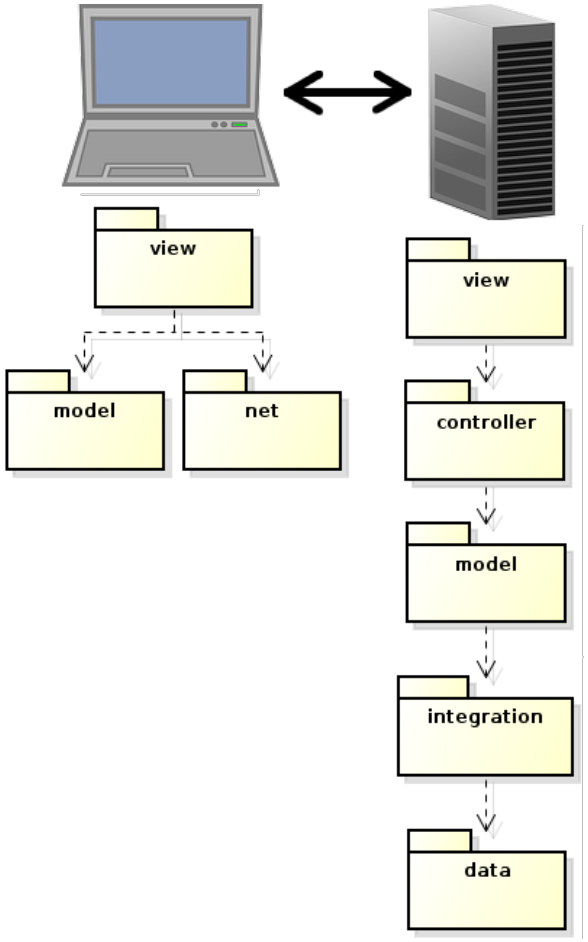
- The server has the same layers as a stand-alone MVC architecture.

Remember: Server-Side Layers



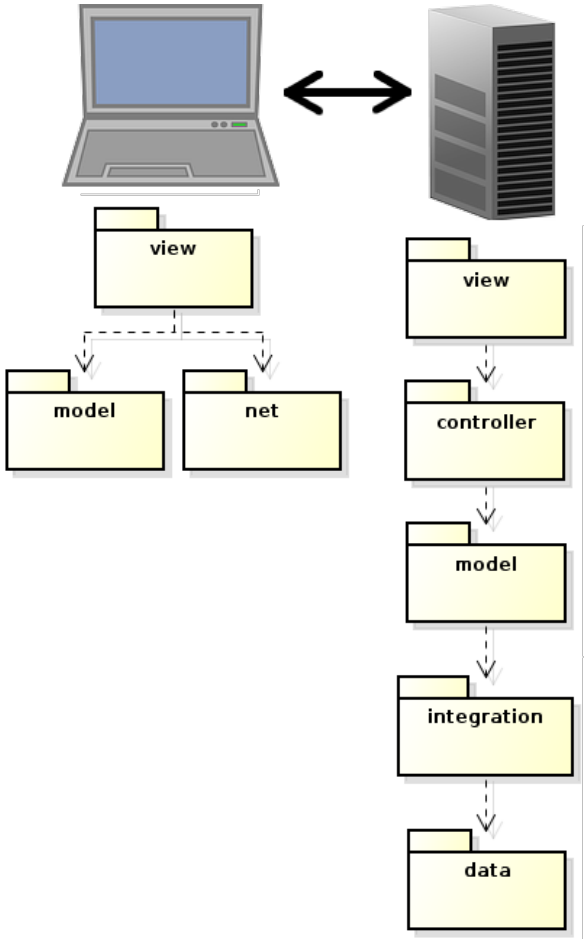
- The server has the same layers as a stand-alone MVC architecture.
- The server's view layer gets HTTP requests and creates HTTP responses.

Remember: Server-Side Layers



- The server has the same layers as a stand-alone MVC architecture.
- The server's view layer gets HTTP requests and creates HTTP responses.
- The MVC pattern states that all UI related code shall be in the view. From controller and down there is only plain object-oriented code.

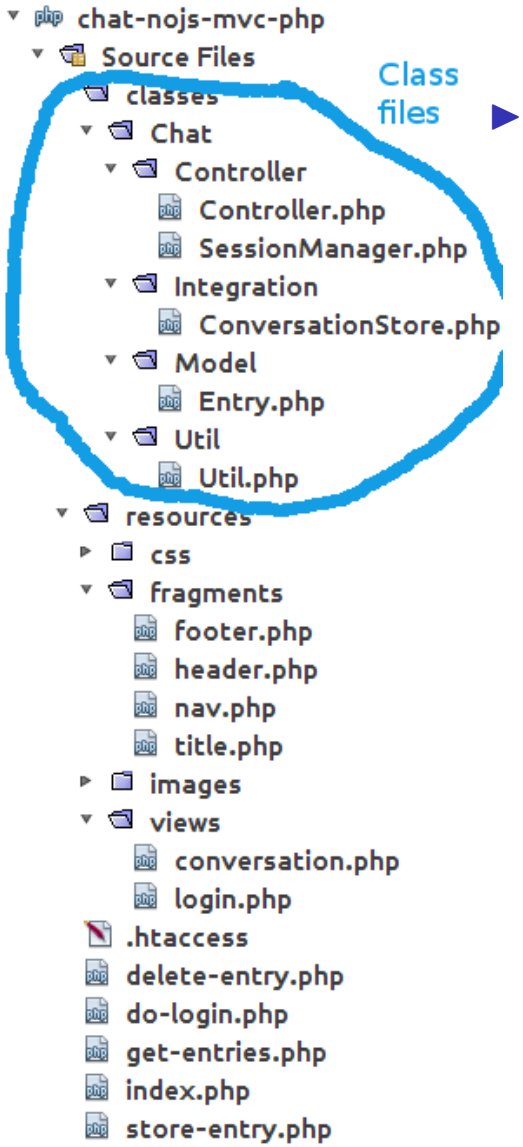
Remember: Server-Side Layers



- The server has the same layers as a stand-alone MVC architecture.
- The server's view layer gets HTTP requests and creates HTTP responses.
- The MVC pattern states that all UI related code shall be in the view. From controller and down there is only plain object-oriented code.

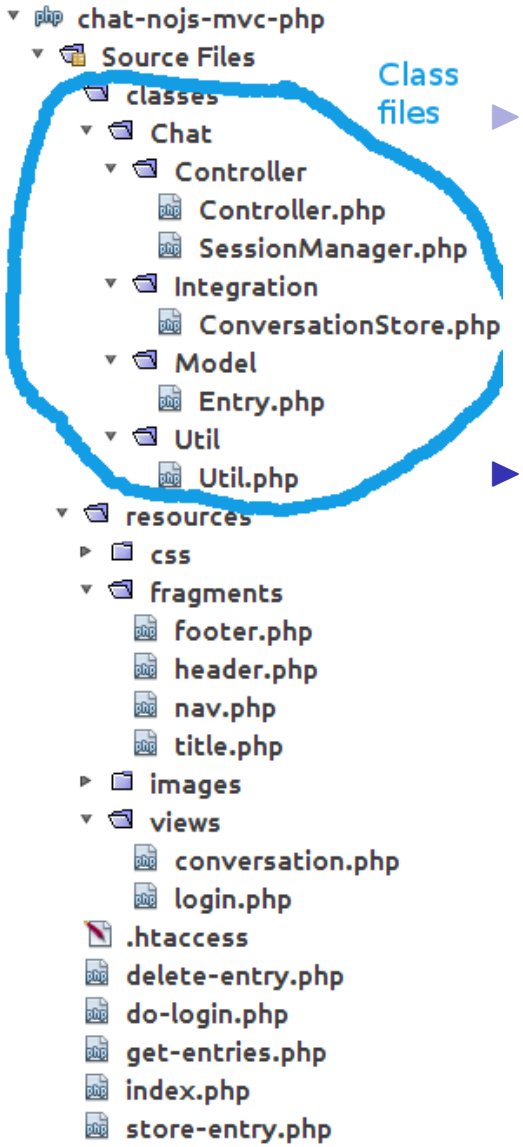
- This means that controller and lower layers are coded exactly as for a stand-alone application. Only the view is specific for a web application.

Class Files



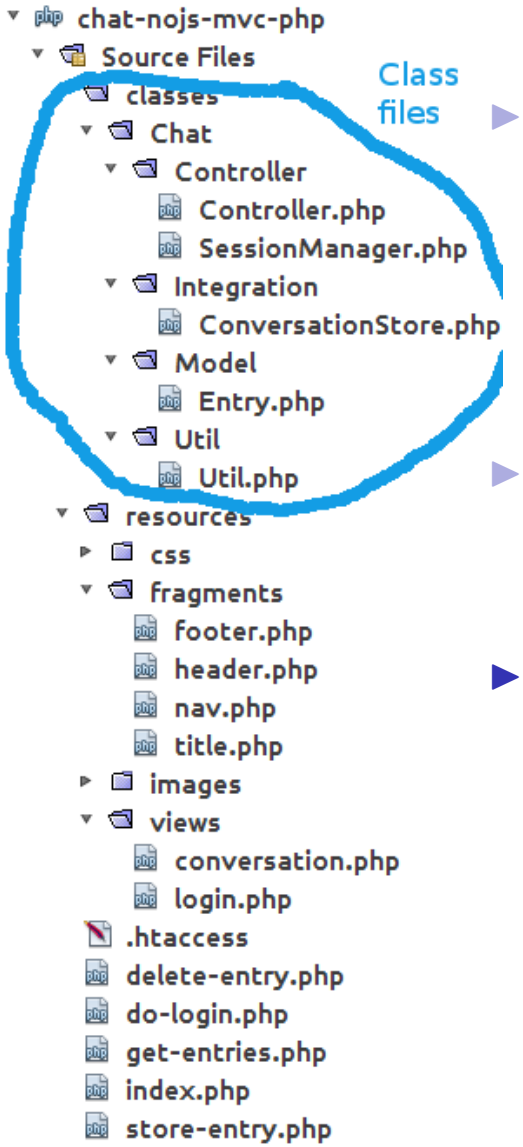
- It is a good practice to organize server-side code as in a Java application. One file per class and one directory per namespace.

Class Files



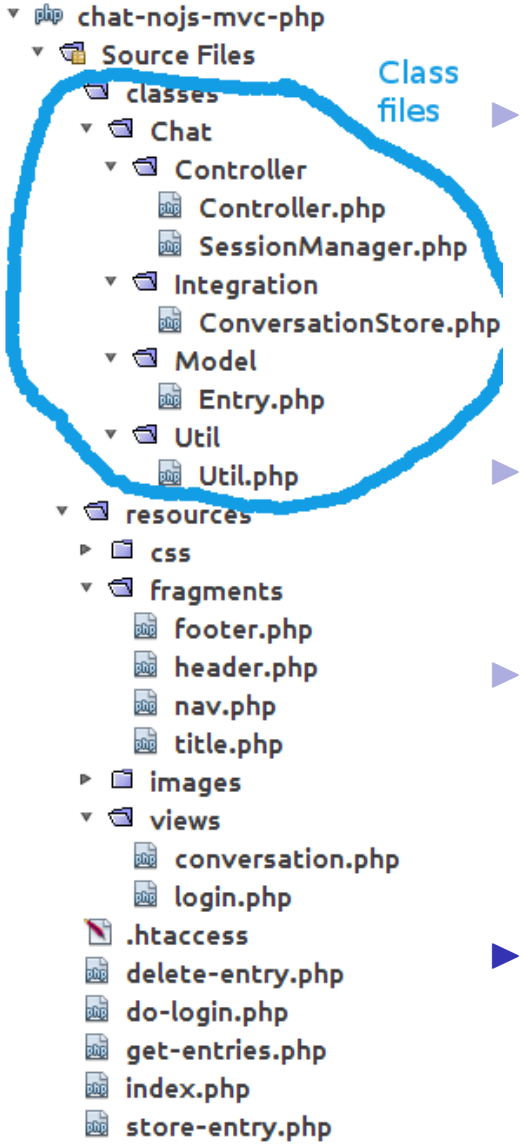
- It is a good practice to organize server-side code as in a Java application. One file per class and one directory per namespace.
- Place all classes in a separate directory, for example **`classes`**.

Class Files



- It is a good practice to organize server-side code as in a Java application. One file per class and one directory per namespace.
- Place all classes in a separate directory, for example **`classes`**.
- Protect classes from direct HTTP access by denying access to the **`classes`** directory.

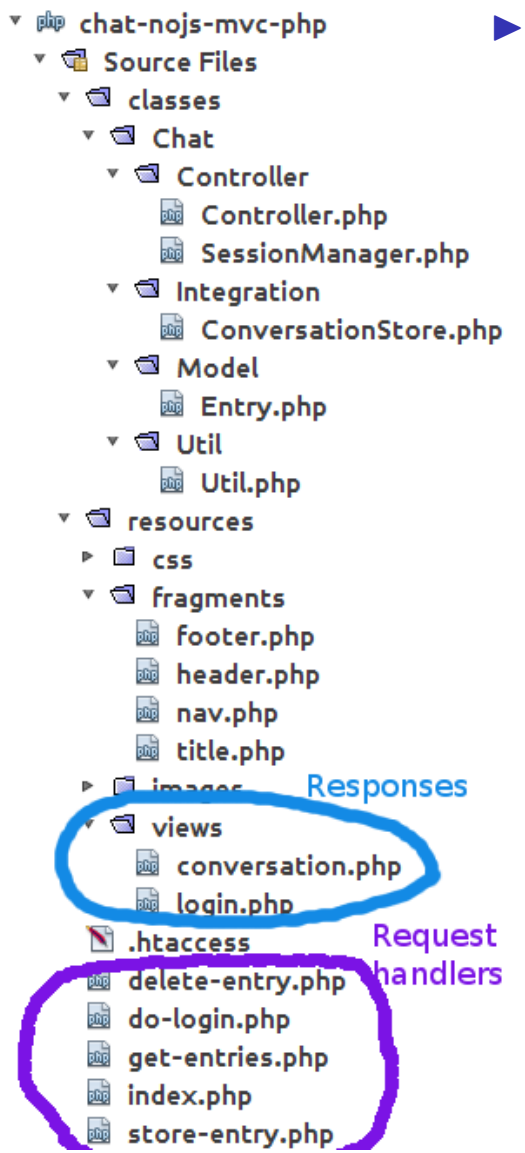
Class Files



- It is a good practice to organize server-side code as in a Java application. One file per class and one directory per namespace.
- Place all classes in a separate directory, for example **`classes`**.
- Protect classes from direct HTTP access by denying access to the **`classes`** directory.
- Enable autoloading classes, see below. This relieves us of **`include`** and **`require`** statements.

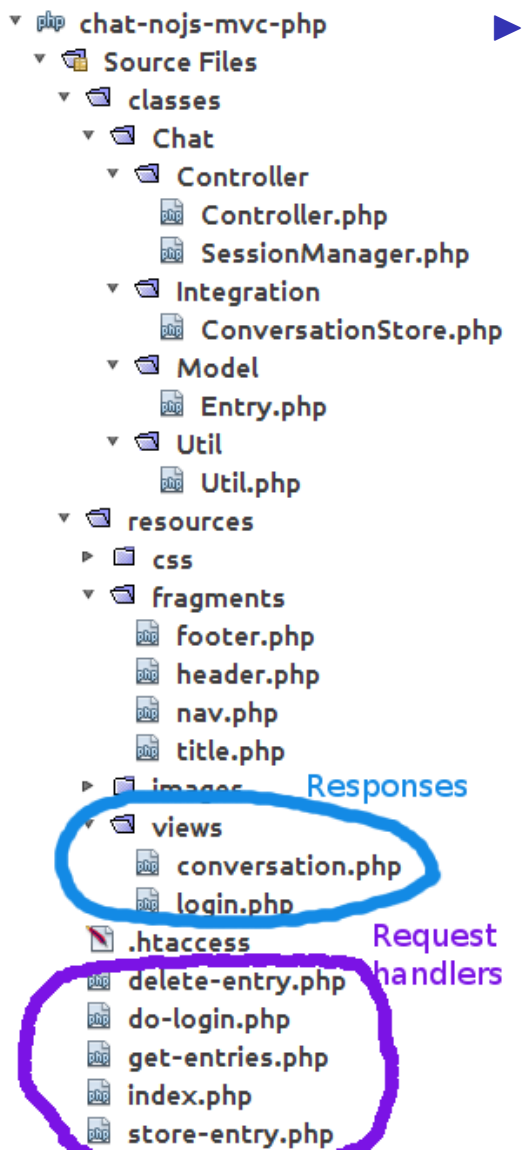
```
spl_autoload_register(function ($class) {
    include 'classes/' . \str_replace('\\', '/', $class) . '.php';
});
```

View Files



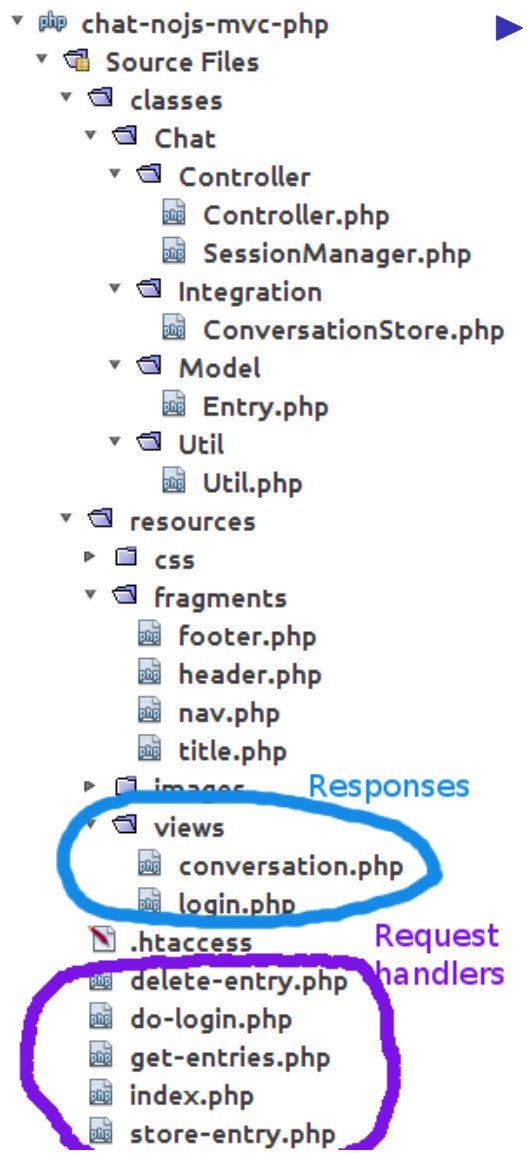
- We would like to place the view in classes. However:

View Files



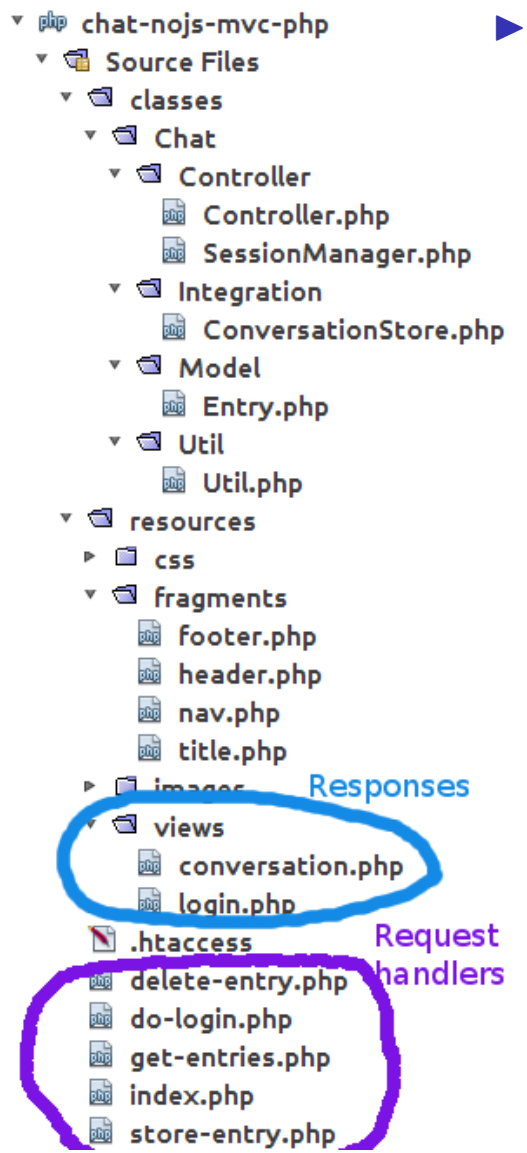
- We would like to place the view in classes. However:
  - We do not want HTML in our PHP classes.

View Files



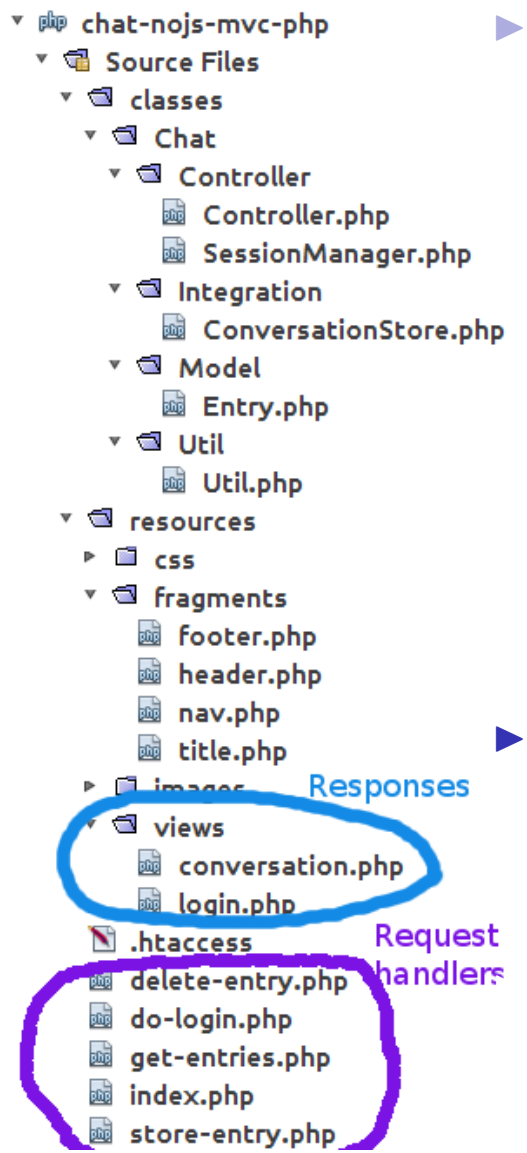
- We would like to place the view in classes. However:
  - We do not want HTML in our PHP classes.
  - We do not want HTTP access to our classes directory.

View Files



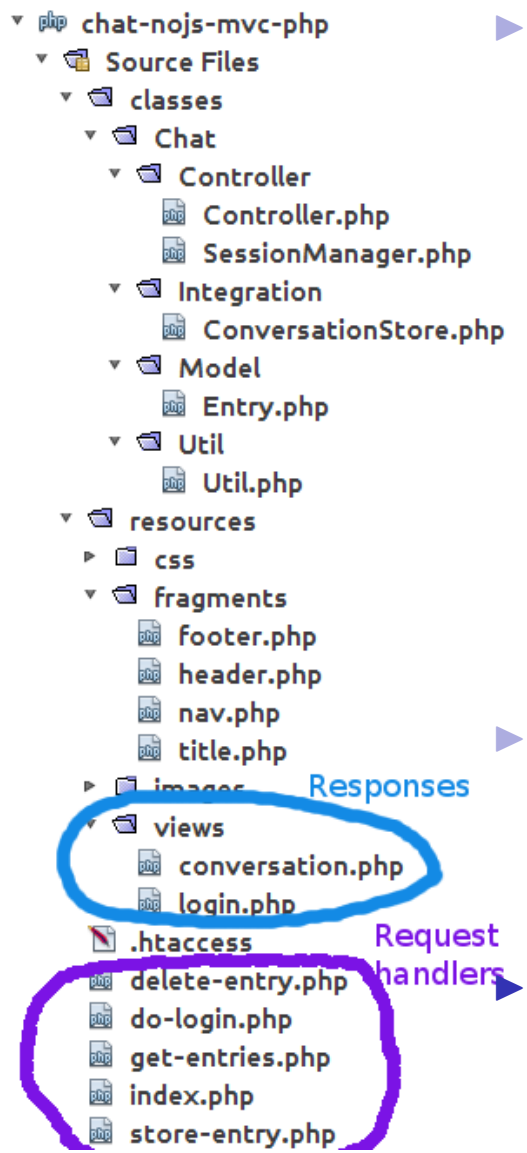
- We would like to place the view in classes. However:
  - We do not want HTML in our PHP classes.
  - We do not want HTTP access to our classes directory.
  - We can not write a URL that addresses a method in a class. A URL can only address a file.

View Files



- We would like to place the view in classes. However:
  - We do not want HTML in our PHP classes.
  - We do not want HTTP access to our classes directory.
  - We can not write a URL that addresses a method in a class. A URL can only address a file.
- Therefore, we need a PHP file without classes to interpret the HTTP request and direct it to the correct classes.

View Files



- We would like to place the view in classes. However:
  - We do not want HTML in our PHP classes.
  - We do not want HTTP access to our classes directory.
  - We can not write a URL that addresses a method in a class. A URL can only address a file.
- Therefore, we need a PHP file without classes to interpret the HTTP request and direct it to the correct classes.
- If the response is a HTML document, we also need to include a HTML file, since we do not want to mix the HTML document with the PHP classes.

Warning: Infrastructure Code!

- There will be quite a lot of code that is identical for each application, for example to:

Warning: Infrastructure Code!

- There will be quite a lot of code that is identical for each application, for example to:
  - Include class files (load classes).

Warning: Infrastructure Code!

- There will be quite a lot of code that is identical for each application, for example to:
  - Include class files (load classes).
  - Route a HTTP request to a method in a class.

Warning: Infrastructure Code!

- There will be quite a lot of code that is identical for each application, for example to:
  - Include class files (load classes).
  - Route a HTTP request to a method in a class.
  - Read HTTP parameters.

Warning: Infrastructure Code!

- There will be quite a lot of code that is identical for each application, for example to:
  - Include class files (load classes).
  - Route a HTTP request to a method in a class.
  - Read HTTP parameters.
  - Include the file with the next view.

Warning: Infrastructure Code!

- There will be quite a lot of code that is identical for each application, for example to:
  - Include class files (load classes).
  - Route a HTTP request to a method in a class.
  - Read HTTP parameters.
  - Include the file with the next view.
  - Include fragments (header, footer, etc) in the view.

Warning: Infrastructure Code!

- There will be quite a lot of code that is identical for each application, for example to:
  - Include class files (load classes).
  - Route a HTTP request to a method in a class.
  - Read HTTP parameters.
  - Include the file with the next view.
  - Include fragments (header, footer, etc) in the view.
- This is called infrastructure code and is a strong call for a framework.

We Must Use a Framework

- A framework is necessary to:

We Must Use a Framework

- A framework is necessary to:
  - Reuse code from previous applications.

We Must Use a Framework

- A framework is necessary to:
  - Reuse code from previous applications.
  - Avoid the big risk of bad architecture.

We Must Use a Framework

- A framework is necessary to:
  - Reuse code from previous applications.
  - Avoid the big risk of bad architecture.
  - Avoid writing new code which means introducing new bugs.

We Must Use a Framework

- A framework is necessary to:
  - Reuse code from previous applications.
  - Avoid the big risk of bad architecture.
  - Avoid writing new code which means introducing new bugs.
  - Thoroughly tested and proven to work well.

We Must Use a Framework

- A framework is necessary to:
  - Reuse code from previous applications.
  - Avoid the big risk of bad architecture.
  - Avoid writing new code which means introducing new bugs.
  - Thoroughly tested and proven to work well.
  - Lots of documentation, easy to get help.

We Must Use a Framework

- A framework is necessary to:
  - Reuse code from previous applications.
  - Avoid the big risk of bad architecture.
  - Avoid writing new code which means introducing new bugs.
  - Thoroughly tested and proven to work well.
  - Lots of documentation, easy to get help.
  - Infrastructure code is difficult to write.

We Must Use a Framework

- A framework is necessary to:
  - Reuse code from previous applications.
  - Avoid the big risk of bad architecture.
  - Avoid writing new code which means introducing new bugs.
  - Thoroughly tested and proven to work well.
  - Lots of documentation, easy to get help.
  - Infrastructure code is difficult to write.
  - Preferably, the framework should use callbacks, i.e., the framework calls our code. Thus, the framework also handles flow control.

Exactly What is the Framework's Task?

- First, we will look at the chat application without a framework, to get a feeling for what is needed.

Exactly What is the Framework's Task?

- First, we will look at the chat application without a framework, to get a feeling for what is needed.
  - We will look at a sample request, namely to write a new entry in the conversation.

Exactly What is the Framework's Task?

- First, we will look at the chat application without a framework, to get a feeling for what is needed.
  - We will look at a sample request, namely to write a new entry in the conversation.
- Then, we will identify what we need the framework to do.

Exactly What is the Framework's Task?

- First, we will look at the chat application without a framework, to get a feeling for what is needed.
  - We will look at a sample request, namely to write a new entry in the conversation.
- Then, we will identify what we need the framework to do.
- Third, we will look at the chat with a framework.

New Entry, `store-entry.php`

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- The HTML form with the new entry is submitted to **`store-entry.php`**

New Entry, `store-entry.php`

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- The HTML form with the new entry is submitted to `store-entry.php`
- Line 6 loads the `Util` class. Since the autoloader is not yet registered, it is loaded manually.

New Entry, `store-entry.php`

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- The HTML form with the new entry is submitted to `store-entry.php`
- Line 6 loads the `Util` class. Since the autoloader is not yet registered, it is loaded manually.
- Line 7 calls the **`initRequest`** method, which performs tasks similar for all requests.

store-entry.php (Cont'd)

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- Lines 9-13 sets **`$msg`** to the value of the HTTP parameter with the new entry. If there is no such parameter, it is set to the empty string.

store-entry.php (Cont'd)

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- Lines 9-13 sets `$msg` to the value of the HTTP parameter with the new entry. If there is no such parameter, it is set to the empty string.
- Line 15 gets the controller of the current session. Remember that all state is lost after a request. We have to store the controller, with its references to the model, in the session.

store-entry.php (Cont'd)

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- Line 16 is the method call to the controller. This is where all request handling is done, the new entry is stored.

store-entry.php (Cont'd)

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- Line 16 is the method call to the controller. This is where all request handling is done, the new entry is stored.
- Lines 17-18 calls the controller to get data that is needed in the next view.

store-entry.php (Cont'd)

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- Line 16 is the method call to the controller. This is where all request handling is done, the new entry is stored.
- Lines 17-18 calls the controller to get data that is needed in the next view.
- Line 19 again stores the controller in the session, for use in the next request.

store-entry.php (Cont'd)

```
1  namespace Chat\View;
2  use \Chat\Util\Util;
3  use \Chat\Model\Entry;
4  use Chat\Controller\SessionManager;
5
6  require_once 'classes/Chat/Util/Util.php';
7  Util::initRequest();
8
9  if (empty($_POST[CHAT_MSG_KEY])) {
10     $msg = "";
11 } else {
12     $msg = $_POST[CHAT_MSG_KEY];
13 }
14
15 $controller = SessionManager::getController();
16 $controller->addEntry(new Entry($controller->getUsername(), $msg));
17 $entries = $controller->getConversation();
18 $username = $controller->getUsername();
19 SessionManager::storeController($controller);
20
21 include CHAT_VIEWS . 'conversation.php';
```

- Line 21 includes the file with the next view. Note that the variables **`$entries`** and **`$username`** are available in that file.

Util.php

```
1  public static function initRequest() {
2      spl_autoload_register(function ($class) {
3          require_once 'classes/' .
4                       \str_replace('\\', '/', $class) .
5                       '.php';
6      });
7
8      session_start();
9      self::defineConstants();
10 }
```

- Lines 2-6 registers the autoloader.

Util.php

```
1  public static function initRequest() {
2      spl_autoload_register(function ($class) {
3          require_once 'classes/' .
4                       \str_replace('\\', '/', $class) .
5                       '.php';
6      });
7
8      session_start();
9      self::defineConstants();
10 }
```

- Lines 2-6 registers the autoloader.
- Line 8 starts a session if there is none.

Util.php

```
1  public static function initRequest() {
2      spl_autoload_register(function ($class) {
3          require_once 'classes/' .
4                       \str_replace('\\', '/', $class) .
5                       '.php';
6      });
7
8      session_start();
9      self::defineConstants();
10 }
```

- Lines 2-6 registers the autoloader.
- Line 8 starts a session if there is none.
- Line 9 creates constants for HTTP parameter keys:

```
1  const SYMBOL_PREFIX = "CHAT_";
2  private static function defineConstants() {
3      self::defineConstant('MSG_KEY', 'msg');
4      self::defineConstant('NICK_KEY', 'nickName');
5      self::defineConstant('TIMESTAMP_KEY', 'timestamp');
6      self::defineConstant('VIEWS', 'resources/views/');
7      self::defineConstant('FRAGMENTS', 'resources/fragments/');
8  }
9  private static function defineConstant($param, $value) {
10     define(self::SYMBOL_PREFIX . $param, $value);
```

SessionManager.php

```
1  const CONTROLLER_KEY = 'controller';
2
3  public static function getController() {
4      if (isset($_SESSION[self::CONTROLLER_KEY])) {
5          return unserialize($_SESSION[self::CONTROLLER_KEY]);
6      } else {
7          return new Controller();
8      }
9  }
10
11 public static function storeController(Controller $controller) {
12     $_SESSION[self::CONTROLLER_KEY] = serialize($controller);
13 }
```

- Line 4 checks if a **`Controller`** object is stored in the session.

SessionManager.php

```
1  const CONTROLLER_KEY = 'controller';
2
3  public static function getController() {
4      if (isset($_SESSION[self::CONTROLLER_KEY])) {
5          return unserialize($_SESSION[self::CONTROLLER_KEY]);
6      } else {
7          return new Controller();
8      }
9  }
10
11 public static function storeController(Controller $controller) {
12     $_SESSION[self::CONTROLLER_KEY] = serialize($controller);
13 }
```

- Line 4 checks if a **`Controller`** object is stored in the session.
- Line 5 reads the stored **`Controller`**.

SessionManager.php

```
1  const CONTROLLER_KEY = 'controller';
2
3  public static function getController() {
4      if (isset($_SESSION[self::CONTROLLER_KEY])) {
5          return unserialize($_SESSION[self::CONTROLLER_KEY]);
6      } else {
7          return new Controller();
8      }
9  }
10
11 public static function storeController(Controller $controller) {
12     $_SESSION[self::CONTROLLER_KEY] = serialize($controller);
13 }
```

- Line 4 checks if a **`Controller`** object is stored in the session.
- Line 5 reads the stored **`Controller`**.
- Line 7 creates a new **`Controller`**.

SessionManager.php

```
1  const CONTROLLER_KEY = 'controller';
2
3  public static function getController() {
4      if (isset($_SESSION[self::CONTROLLER_KEY])) {
5          return unserialize($_SESSION[self::CONTROLLER_KEY]);
6      } else {
7          return new Controller();
8      }
9  }
10
11 public static function storeController(Controller $controller) {
12     $_SESSION[self::CONTROLLER_KEY] = serialize($controller);
13 }
```

- Line 4 checks if a **Controller** object is stored in the session.
- Line 5 reads the stored **Controller**.
- Line 7 creates a new **Controller**.
- Line 12 stores the **Controller** in the session.

The view, `conversation.php`

- The view should consist of only HTML.
  Unfortunately, this goal is not reached:

The view, `conversation.php`

- The view should consist of only HTML. Unfortunately, this goal is not reached:
- First, since there are header, footer and navigation fragments that appear on each page, we have to include them to avoid duplicated code. These inclusions are PHP statements, see lines 2 and 6 below.

```
1  ...
2  <header class="section group">
3      <?php include CHAT_FRAGMENTS . 'header.php' ?>
4  </header>
5
6  <main class="section group">
7      <nav class="section group">
8          <?php include CHAT_FRAGMENTS . 'nav.php' ?>
9      </nav>
10 ...
```

The view (Cont'd)

- Second, to generate the conversation view from the **`$entries`** variable is also PHP code.

```
1   ...
2 <div class="col span_4_of_4">
3     <?php
4     foreach ($entries as $entry) {
5         if (!$entry->isDeleted()) {
6             echo ("<p class='author'>" . $entry->getNickName() . ":</p>")
7             echo("<p class='entry'>");
8             echo(nl2br($entry->getMsg()));
9             echo ("</p>");
10            if ($entry->getNickName() === $username) {
11                echo("<form action='delete-entry.php'>");
12                echo("<input type='hidden' name='timestamp' value='" .
13                $entry->getTimestamp() . "'/>");
14                echo("<input type='submit' value='Delete'/>");
15                echo("</form>");
16            }
17        }
18    }
19    ?>
20 </div>
21   ...
```

Other Layers, No Problem

- Now we have seen all view code related to creating a new entry in the conversation. The view is normally the hardest part of a web application.

Other Layers, No Problem

- Now we have seen all view code related to creating a new entry in the conversation. The view is normally the hardest part of a web application.
- Controller and lower layers are plain PHP code, created with normal object-oriented analysis, design and programming methodologies.

Question 1

Let's Look for Infrastructure Code

- In **`store-entry`**, **`Util`** and **`SessionManager`** there is no code at all specific for this application!

Let's Look for Infrastructure Code

- In **`store-entry`**, **`Util`** and **`SessionManager`** there is no code at all specific for this application!
- One could argue that the call to the controller in **`store-entry.php`** is application specific.

Let's Look for Infrastructure Code

- In **`store-entry`**, **`Util`** and **`SessionManager`** there is no code at all specific for this application!
- One could argue that the call to the controller in **`store-entry.php`** is application specific.
  - However, we are rid of also this line if the framework allows us to specify a URL-to-method mapping, which most frameworks do.

Let's Look for Infrastructure Code

- In `store-entry`, `Util` and `SessionManager` there is no code at all specific for this application!
- One could argue that the call to the controller in `store-entry.php` is application specific.
  - However, we are rid of also this line if the framework allows us to specify a URL-to-method mapping, which most frameworks do.
- One could also argue that the names of the HTTP parameters are application specific.

Let's Look for Infrastructure Code

- In `store-entry`, `Util` and `SessionManager` there is no code at all specific for this application!
- One could argue that the call to the controller in `store-entry.php` is application specific.
  - However, we are rid of also this line if the framework allows us to specify a URL-to-method mapping, which most frameworks do.
- One could also argue that the names of the HTTP parameters are application specific.
  - But, most frameworks enable specifying those as method parameters in the URL-to-method-mapping!

The Framework's tasks

- Therefore, the framework must handle:

The Framework's tasks

- Therefore, the framework must handle:
  - Class loading, i.e., include PHP class files.

The Framework's tasks

- Therefore, the framework must handle:
  - Class loading, i.e., include PHP class files.
  - Routing, which means to map a URL to a specified method in a specified class.

The Framework's tasks

- Therefore, the framework must handle:
  - Class loading, i.e., include PHP class files.
  - Routing, which means to map a URL to a specified method in a specified class.
  - HTTP parameters, it should be possible to specify how parameters are passed as arguments to the methods specified by the routing rules.

The Framework's tasks

- Therefore, the framework must handle:
  - Class loading, i.e., include PHP class files.
  - Routing, which means to map a URL to a specified method in a specified class.
  - HTTP parameters, it should be possible to specify how parameters are passed as arguments to the methods specified by the routing rules.
  - HTTP sessions, all objects in controller and lower layers should be stored in the **`$_SESSION`** superglobal.

The Framework's tasks

- Therefore, the framework must handle:
  - Class loading, i.e., include PHP class files.
  - Routing, which means to map a URL to a specified method in a specified class.
  - HTTP parameters, it should be possible to specify how parameters are passed as arguments to the methods specified by the routing rules.
  - HTTP sessions, all objects in controller and lower layers should be stored in the `$_SESSION` superglobal.
  - Templating, to generate a view from data, we need something to replace the PHP code looping through the **`$conversation`** variable.

The Framework's tasks, Cont'd

- The framework must handle:

The Framework's tasks, Cont'd

- The framework must handle:
  - Composite views, there should be a mechanism to specify fragments (header, footer etc) for inclusion without having to mix HTML and PHP.

The Framework's tasks, Cont'd

- The framework must handle:
  - Composite views, there should be a mechanism to specify fragments (header, footer etc) for inclusion without having to mix HTML and PHP.
  - Not only should it be possible to reuse the fragments, also the page layout should be reused. This means only the content of the main area should be specific for a page.

The Framework's tasks, Cont'd

- The framework must handle:
  - Composite views, there should be a mechanism to specify fragments (header, footer etc) for inclusion without having to mix HTML and PHP.
  - Not only should it be possible to reuse the fragments, also the page layout should be reused. This means only the content of the main area should be specific for a page.
- There are many other requirements that should be managed by a framework, but which we have skipped in this small example.

Section

- MVC in a PHP Web Application
- The id1354-fw Framework

PHP Frameworks

- There are **many** PHP frameworks, of different size and quality.

PHP Frameworks

- There are **many** PHP frameworks, of different size and quality.
- Some interesting and often used frameworks are Zend, Symfony, Yii, Laravel and Phalcon.

PHP Frameworks

- There are **many** PHP frameworks, of different size and quality.
- Some interesting and often used frameworks are Zend, Symfony, Yii, Laravel and Phalcon.
- Here, we will have a look at a framework written specifically for this course, the id1354-fw framework.

The id1354-fw Framework

- But frameworks should be reused?? Why write a new one?

The id1354-fw Framework

- But frameworks should be reused?? Why write a new one?
- The most common frameworks are too complicated for this course. Smaller frameworks are often unstable, lack documentation, and solve wrong problems.

The id1354-fw Framework

- But frameworks should be reused?? Why write a new one?
- The most common frameworks are too complicated for this course. Smaller frameworks are often unstable, lack documentation, and solve wrong problems.
- The id1354-fw framework is very small, but still has exactly the features we are looking for (except templating and composite views) and nothing more. It will also be supported as long as it is used in the course.

The id1354-fw Framework

- But frameworks should be reused?? Why write a new one?
- The most common frameworks are too complicated for this course. Smaller frameworks are often unstable, lack documentation, and solve wrong problems.
- The id1354-fw framework is very small, but still has exactly the features we are looking for (except templating and composite views) and nothing more. It will also be supported as long as it is used in the course.
- Now, we will look at the id1354-fw framework and how it changes the Chat application.

The id1354-fw Framework

- But frameworks should be reused?? Why write a new one?
- The most common frameworks are too complicated for this course. Smaller frameworks are often unstable, lack documentation, and solve wrong problems.
- The id1354-fw framework is very small, but still has exactly the features we are looking for (except templating and composite views) and nothing more. It will also be supported as long as it is used in the course.
- Now, we will look at the id1354-fw framework and how it changes the Chat application.
- The full documentation, including installation instructions, is included in the **`id1354-fw.zip`** file available at the course web.

Class Loading

- You do not have to **`include`** or **`require`** any classes, they are loaded by the framework.

Class Loading

- You do not have to **`include`** or **`require`** any classes, they are loaded by the framework.
- Place all your classes under the **`classes`** directory that is created when the framework is installed.

Class Loading

- You do not have to **`include`** or **`require`** any classes, they are loaded by the framework.
- Place all your classes under the **`classes`** directory that is created when the framework is installed.
- Use a directory structure matching the namespaces and name each file after the class in the file.

Class Loading

- You do not have to **`include`** or **`require`** any classes, they are loaded by the framework.
- Place all your classes under the **`classes`** directory that is created when the framework is installed.
- Use a directory structure matching the namespaces and name each file after the class in the file.
  - For example, the class **`MyClass`** in the namespace **`\MyApp\Model`** shall be in the file **`classes/MyApp/Model/MyClass.php`**.

Routing

- To create a class that handles a HTTP request, write a class that extends **`\Id1354fw\View\AbstractRequestHandler`**.

Routing

- To create a class that handles a HTTP request, write a class that extends `\Id1354fw\View\AbstractRequestHandler`.
- This class shall have the method **`protected function doExecute()`**, which performs all work needed to handle the http request.

Something.

Routing

- To create a class that handles a HTTP request, write a class that extends `\Id1354fw\View\AbstractRequestHandler`.
- This class shall have the method `protected function doExecute()`, which performs all work needed to handle the http request.
- If this class is called `\MyApp\View\Something`, the doExecute method is called when the user requests the url `http://<yourserver>/<yourwebapp>/Myapp/View/Something`.

HTTP Parameters

- The HTTP request handling class must have a set method for each http post and get parameter.

HTTP Parameters

- The HTTP request handling class must have a set method for each http post and get parameter.
- If the parameter is called **`myParam`**, the set method must be
  `public function setMyParam($value)`.

HTTP Parameters

- The HTTP request handling class must have a set method for each http post and get parameter.
- If the parameter is called **`myParam`**, the set method must be `public function setMyParam($value)`.
- This function will be called with the value of the http parameter before the **`doExecute`** method is called.

Question 2

Sessions

- The request handler contains the object **`$this->session`**, which has the following session handling methods.

Sessions

- The request handler contains the object **`$this->session`**, which has the following session handling methods.
  - **`restart`**, starts a new session if there is none. Changes session id if there is already a session.

Sessions

- The request handler contains the object **`$this->session`**, which has the following session handling methods.
  - **`restart`**, starts a new session if there is none. Changes session id if there is already a session.
  - **`invalidate`**, stops the session, discards all session data, unsets the session id and destroys the session cookie.

Sessions

- The request handler contains the object **`$this->session`**, which has the following session handling methods.
  - **`restart`**, starts a new session if there is none. Changes session id if there is already a session.
  - **`invalidate`**, stops the session, discards all session data, unsets the session id and destroys the session cookie.
  - **`set`** ,stores a key/value pair in the session.

Sessions

- The request handler contains the object **`$this->session`**, which has the following session handling methods.
  - **`restart`**, starts a new session if there is none. Changes session id if there is already a session.
  - **`invalidate`**, stops the session, discards all session data, unsets the session id and destroys the session cookie.
  - **`set`** ,stores a key/value pair in the session.
  - **`get`**, reads a value stored in the session.

View Handling

- To make available data in the next view, call the method **`addVariable($name, $value)`** in the **`doExecute`** method.

View Handling

- To make available data in the next view, call the method `addVariable($name, $value)` in the `doExecute` method.
- This will make the parameter `$value` available in the next view, in a variable called `$name`.

View Handling

- To make available data in the next view, call the method **`addVariable($name, $value)`** in the **`doExecute`** method.
- This will make the parameter **`$value`** available in the next view, in a variable called **`$name`**.
- The **`doExecute`** method shall return the path to the file with the next view. **`views/`** is prepended to the returned path and **`.php`** is appended to the path.

Question 3

Composite Views and Templates

- Unfortunately, composite views and templates are not handled by the id1354-fw framework.

Composite Views and Templates

- Unfortunately, composite views and templates are not handled by the id1354-fw framework.
- Therefore, we still have to mix PHP in the HTML code to include data and fragments in the view.

Mission Completed (Almost)

- Now compare the chat application with and without the framework. With the framework, there is no infrastructure code!!

Mission Completed (Almost)

- Now compare the chat application with and without the framework. With the framework, there is no infrastructure code!!
- Adding more functionality involves only new implementations of **`AbstractRequestHandler`**, and ordinary object-oriented code in controller and lower layers.

Mission Completed (Almost)

- Now compare the chat application with and without the framework. With the framework, there is no infrastructure code!!
- Adding more functionality involves only new implementations of `AbstractRequestHandler`, and ordinary object-oriented code in controller and lower layers.
- All this is application specific!

Mission Completed (Almost)

- Now compare the chat application with and without the framework. With the framework, there is no infrastructure code!!
- Adding more functionality involves only new implementations of `AbstractRequestHandler`, and ordinary object-oriented code in controller and lower layers.
- All this is application specific!
- But there is still PHP in the HTML code...