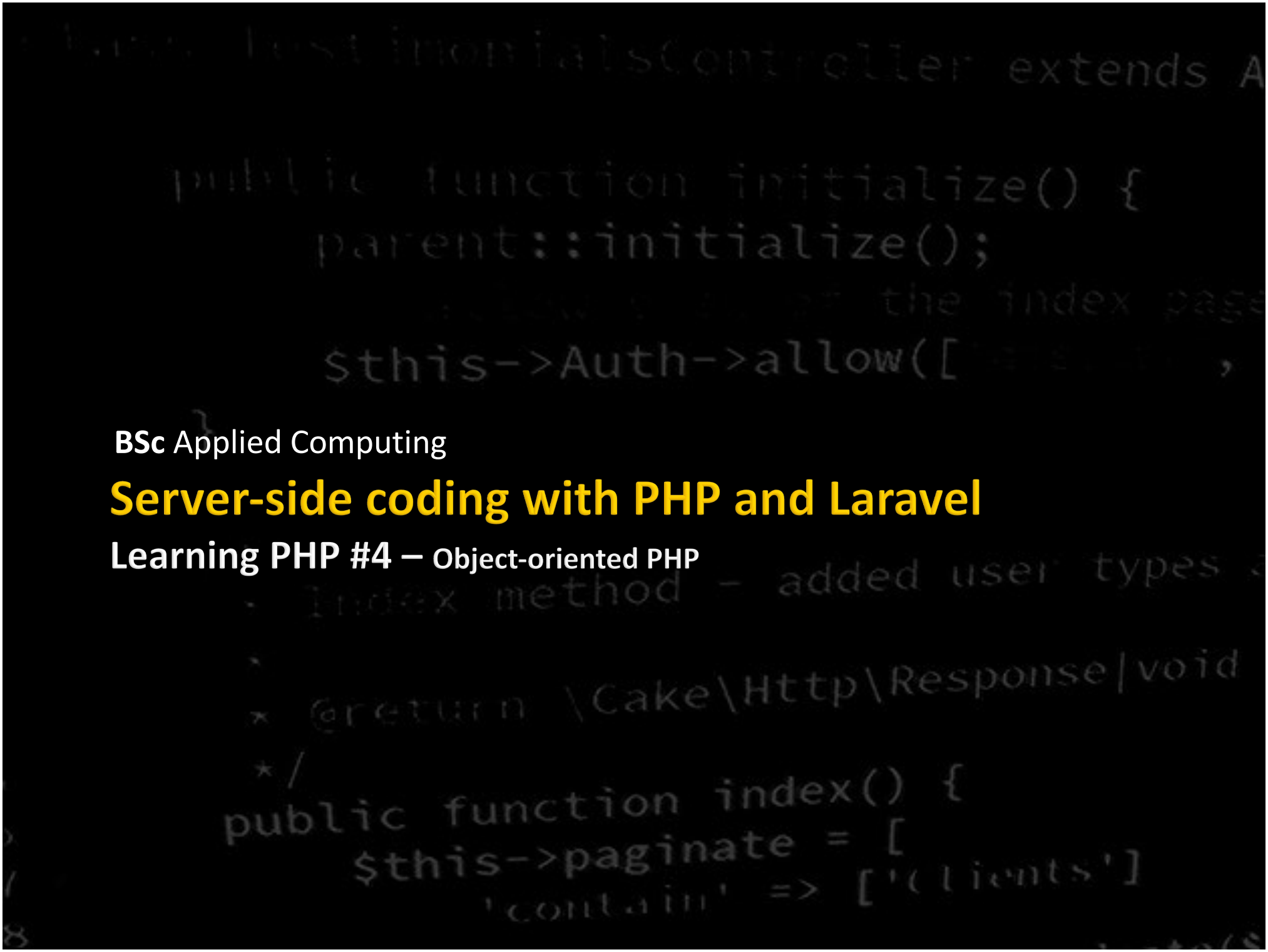


BSc Applied Computing

Server-side coding with PHP and Laravel

Learning PHP #4 – Object-oriented PHP

In this session…

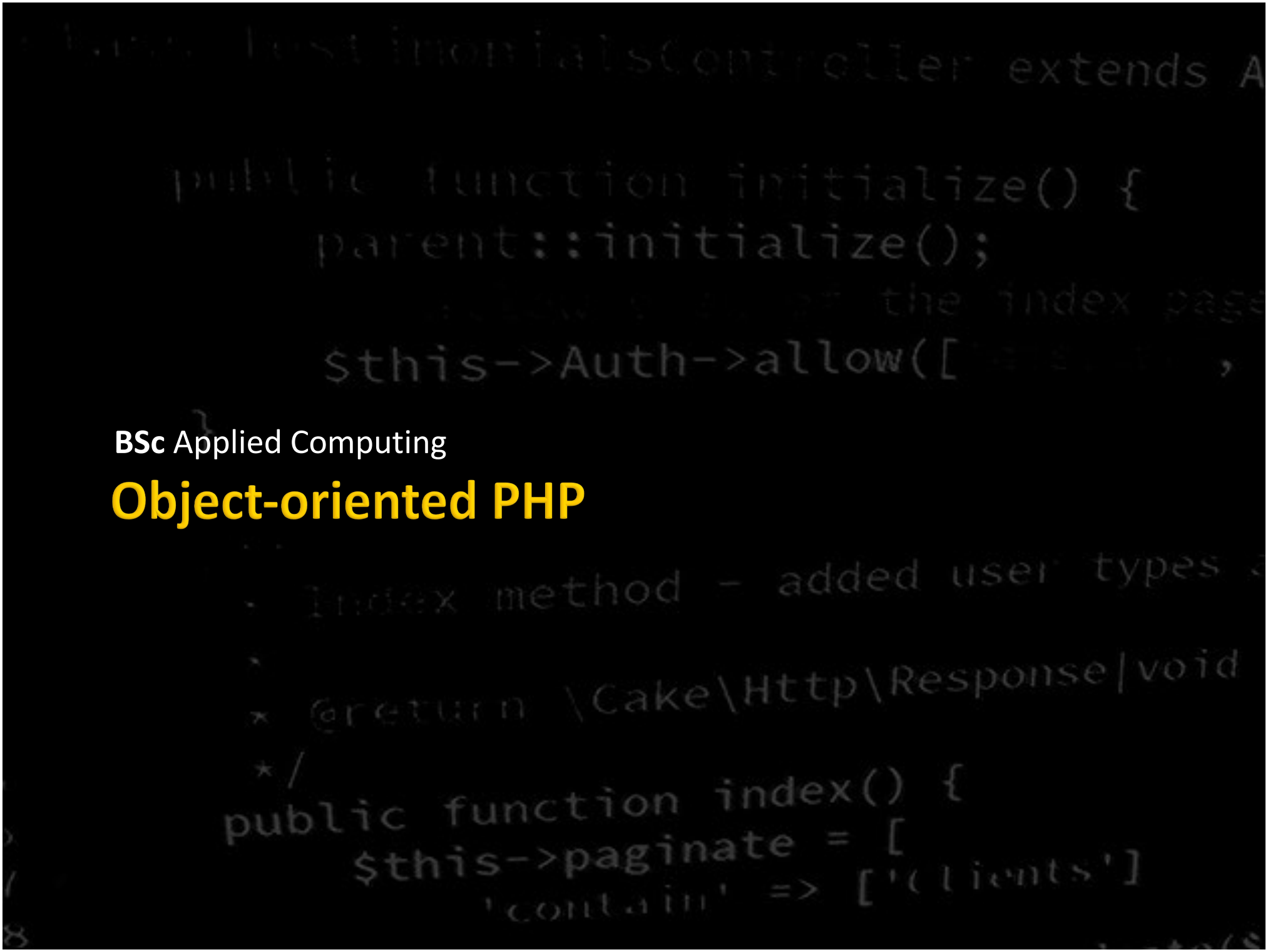
- Recap PHP and MySQL connectivity
  - Last exercise progress
  - Reading list
  - Stateful vs RESTful
  - Use of the PHP session
- PHP and Object-orientation
  - PHP and OOP
  - Namespace and foldering
  - Using a specific class
  - MVC and frameworks
- Simple OOP user account system
  - Project setup
  - Project development workshop

BSc Applied Computing

Recap PHP and MySQL connectivity

Quick recap

- Reasons to use a database:
  - The site's content changes frequently
  - The business uses ecommerce
  - The site employs user-driven content
- PHP can use a lot of databases
- MySQL, a database server
- PHPMyAdmin use
- User account access for the webserver



BSc Applied Computing

Object-oriented PHP

Object orientation

- **“Object-oriented programming** (**OOP**) is a programming paradigm based on the concept of "objects", which can contain data, in the form of fields (often known as *attributes* or *properties*), and code, in the form of procedures (often known as *methods*). A feature of objects is an object's procedures that can access and often modify the data fields of the object with which they are associated (objects have a notion of "this" or "self").

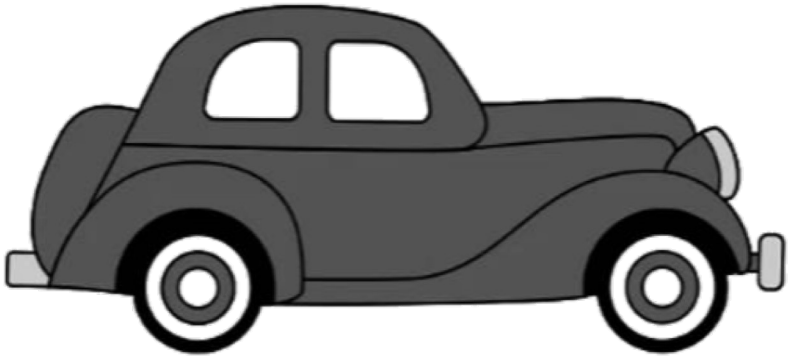
  In OOP, computer programs are designed by making them out of objects that interact with one another. OOP languages are diverse, but the most popular ones are class-based, meaning that objects are instances of classes, which also determine their types. “

  - Wikipedia

Object-oriented programming

OBJECT-ORIENTED PROGRAMMING

Properties
Make
Model
Color
Year
Price

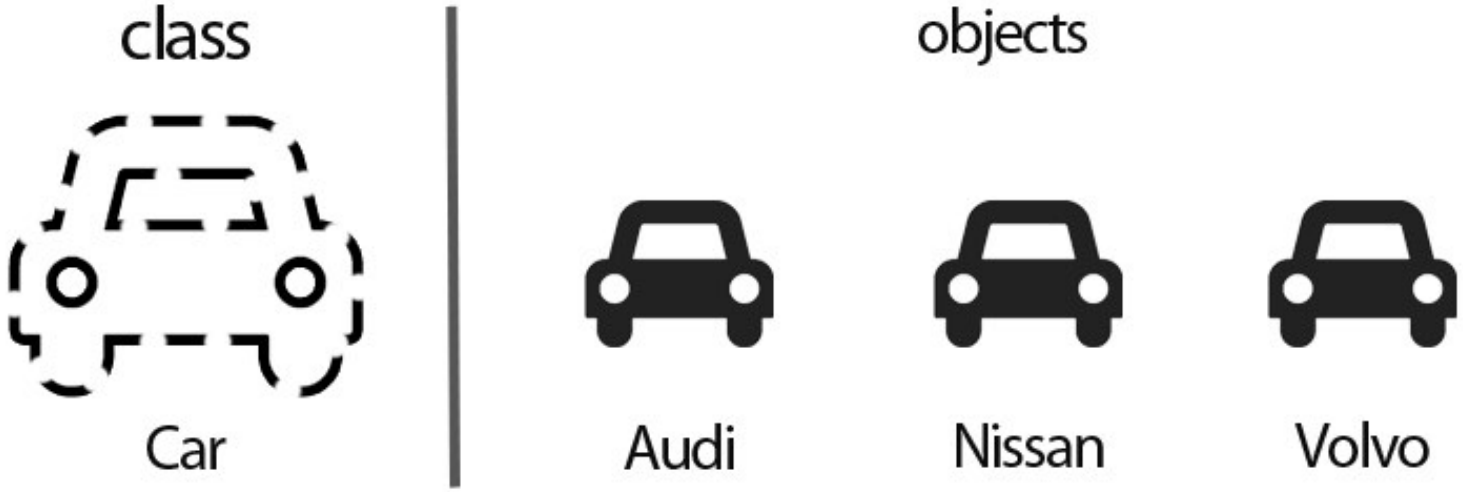


Events
On_Start
On_Parked
On_Brake

Methods
Start
Drive
Park

OOP Rationale

- **Code Reuse and Recycling:** Other projects
- **Encapsulation:** Black-box code
- **Modularity**: More planning but better division
- **Software Maintenance:** Much easier



PHP is not OO by default

- PHP is a procedural language.
- PHP has acquired OOP capabilities as it developed as a language.
- PHP is only OOP if you are coding and working with classes and that classes are the dominant paradigm used to construct the application.
- OOP splits up the application into manageable functional chunks.
- Helps make larger projects more understandable.
- Allows a development team work on different aspects of the same application simultaneously without interfering with each other.
- Most server-side frameworks use the OOP paradigm (Laravel, CakePHP, etc).
- Creating an API using a framework is much easier if you understand OOP.
- Using a framework to create an API is much easier than coding everything all by yourself.

Namespace and foldering

- Large projects can include a lot of classes:
  - Created specifically for the project
  - Framework or library classes for common tasks
- Classes are usually organised as a collection of files, one class per file.
- Example class on the next slide.
- A meaningful folder structure is important when using classes as this will be reflected in the namespace given to any specific class.
- All classes associated with a specific project should have a root folder, often with the filename of “class” or “classes”.
- The folder structure of the class becomes its namespace and this then allows many classes with the same class name to be used together in the same project without fear of collision.

Example class

```php
<? PHP
    // namespace for this class (usually folder structure of the applicaiton)
    namespace classes;

    // class declaration
    class Person {

        // member variables
        private $m_firstName;
        private $m_surName;
        private $m_age;

        /**
         * Constructor function
         */
        public function __construct($fn,$sn,$ag) {
            $this->m_firstName = $fn;
            $this->m_surName = $sn;
            $this->m_age = $ag;
        }

        /**
         * Print object function
         * returns string
         */
        public function toString() {
            return $this->m_firstName . " " . $this->m_surName . " is " . $this->m_age;
        }
    }
?>
```

Using a specific class

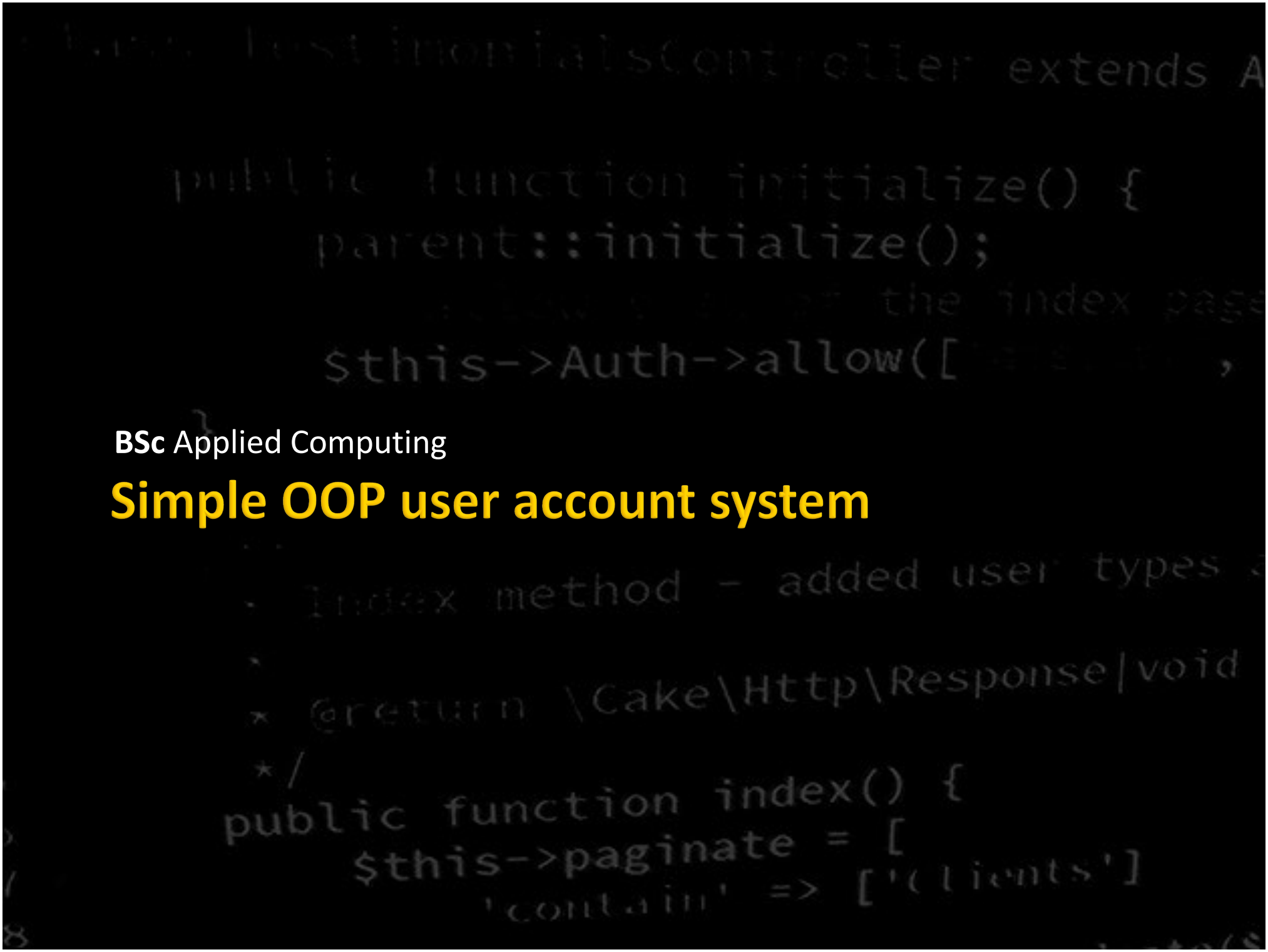
- In order to use any given class in an application, PHP requires the file to be specifically declared, traditionally with a **require** or **include** statement.
- Many classes can be required by some areas of code.
- To speed up the declaration and use of classes PHP programmers use an **auto load call back function** which loads classes in as the programmer tries to use them and the **use** keyword to avoid having to declare the namespace each time a specific object is called.

```
3       // autoload any classes that we attempt to use
4       spl_autoload_register(function($className) {
5           include_once $_SERVER['DOCUMENT_ROOT'] . '/' . $className . '.php';
6       });
7
8       // Include the Person class
9       use classes\Person;
10
11      // instanciate a new object
12      $person = new Person("Andrew","Green",21);
```

MVC and frameworks

- Model – data structure representation of a database record.
- View – the final page sent to the browser.
- Controller – the application logic.

- MVC is a design pattern often used in advanced web application design.
- It is a design pattern employed by many of the popular application. development frameworks (Laravel, CakePHP, etc).
- MVC splits the project into three distinct areas which can then be assigned to different programming teams.
- MVC allows areas of the project to be easily overhauled (for example, if the project moves to a new database or new view framework the application logic does not need any adjustments).


BSc Applied Computing
Simple OOP user account system

Project setup

- You should create a new virtual host for this project (use the instructions from week 1 if needed)
- **Project folder:** UserAccountSystem
- **Project URL:** account.system

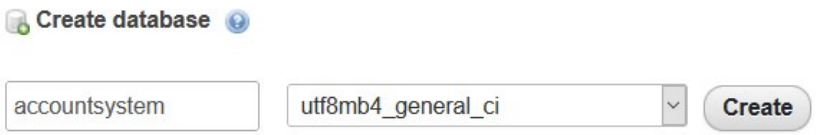
- This tutorial will use the **Brackets** editor.
- **All new code will be highlighted**, other code will be included to indicate placement of new code only and should not be entered.
- PHP can use // or # to indicate a comment.
- This tutorial will use // for all comments.
- Further explanation of code is included in the notes pane of specific slides.

This project is different!

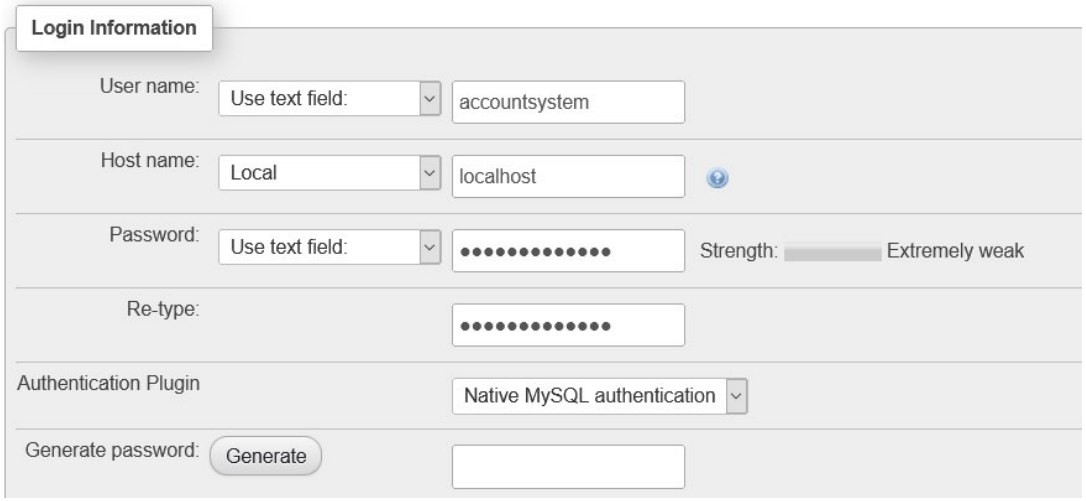
- Up to now we have dealt with smaller, self contained application projects.
- This project is intended to demonstrate one aspect of a much larger system.
- It involves far more coding than previous projects and it is unlikely that you will finish it during the session.
- Make sure that you follow the rest of this tutorial in your own time and complete the project for next week.
- This project contains many important aspects which later sessions will draw upon, continuing and completing this project in your own time is **extremely important** if you want to get the most out of the following sessions.
- I will be reviewing your progress with this application at the start of the next session.

Database setup

- This project will use a database setup script to configure tables required
- The database itself and the user account for the webserver still need to be set up using PHPMyAdmin.*

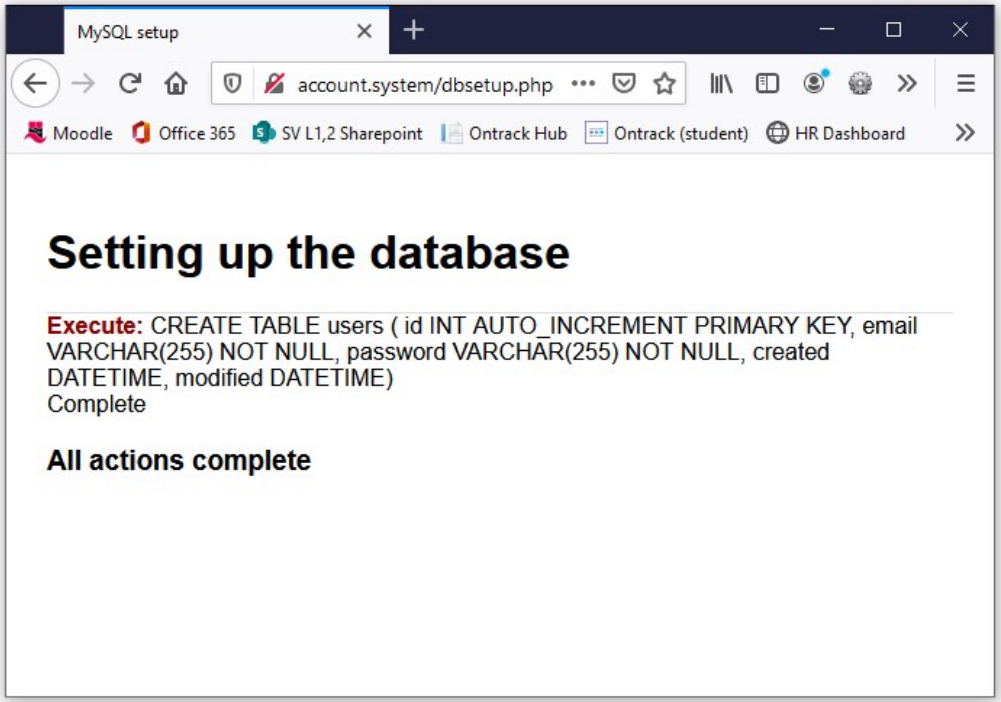


- Set up a new database with the following details:
- **Database:** accountsystem
- **Username:** accountsystem
- **Password:** accountsystem
- **Hostname:** localhost



Database setup

- Download the database setup script (**dbsetup.php**) from Moodle into the **UserAccountSystem** folder.
- Open your browser and in the address bar type **http://account.system/dbsetup.php.**



Database setup

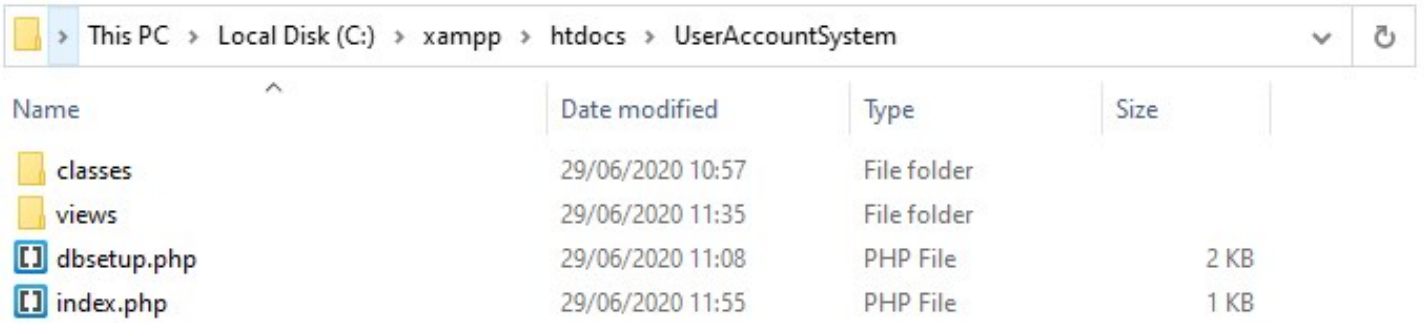
- Double check the database setup by inspecting the new **users** table in **PHPMyAdmin.**

	#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
☐	1	**id**	int(11)			No	*None*		AUTO_INCREMENT	Change Drop More
☐	2	**email**	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
☐	3	**password**	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
☐	4	**created**	datetime			Yes	*NULL*			Change Drop More
☐	5	**modified**	datetime			Yes	*NULL*			Change Drop More

- This is the only table that our simple user account system will need.

Project setup

- Open Brackets and add a new **index.php** file to the **UserAccountSystem** folder as in previous sessions.
- Add two folders to this same **UserAccountSystem** folder:
  - classes
  - views
- The **UserAccountSystem** folder should now look like this.



Coding index.php

- The **index** page will only hold a very rough project framework which will be filled by specific views served from the **views** folder as dictated by a project controller class.
- In Brackets, add a PHP block to the top of the **index.php** page.
- Next add a class autoloader call-back function as shown below.

```php
<?PHP

// autoload any classes that we attempt to use
spl_autoload_register(function($className) {
    include_once $_SERVER['DOCUMENT_ROOT'] . '/' . $className . '.php';
});

?>
```

Coding index.php

- Next, add a **use** statement to include a class called **UserController** (we have not created this yet), which is found in the **classes** folder.
- Thanks to the **autoload** function, any class (either bespoke or third party) named in a **use** statement will be available to the project.
- The **use** statement must be included before any display logic or HTML which is why it is included at the top of this file.

```
1  <?PHP
2
3  // autoload any classes that we attempt to use
4  spl_autoload_register(function($className) {
5      include_once $_SERVER['DOCUMENT_ROOT'] . '/' . $className . '.php';
6  });
7
8  // Includes
9  use classes\UserController;
```

Coding index.php

- Next, add a **use** statement to include a class called **UserController** (we have not created this yet), which is found in the **classes** folder.
- Thanks to the **autoload** function, any class (either bespoke or third party) named in a **use** statement will be available to the project.
- The **use** statement must be included before any display logic or HTML which is why it is included at the top of this file.
- Add the highlighted code.

```
1   <?PHP
2
3   // autoload any classes that we attempt to use
4   spl_autoload_register(function($className) {
5       include_once $_SERVER['DOCUMENT_ROOT'] . '/' . $className . '.php';
6   });
7
8   // Includes
9   use classes\UserController;
10
```

Coding index.php

- This application is a user-login system so the PHP session is required to keep track of users who have successfully logged in.
- Add a **session_start()** statement to indicate that the session will be used.
- Finally, we instantiate the **UserController** object and create a reference for it in **$uc.**
- Add the highlighted code as shown below.

```
 8  // Includes
 9  use classes\UserController;
10
11  // start the session
12  session_start();
13
14  // instantiate a user controller
15  $uc = new UserController();
16
17  ?>
```

Coding index.php

- Next, add the minimal framework HTML to the **index.php** page.
- This contains a few PHP print statements which will include public properties of the **$uc** (**UserController**).
- Add the highlighted code.

```
14  // instantiate a user controller
15  $uc = new UserController();
16
17  ?>
18  <!DOCTYPE html>
19  <html lang="en">
20      <head>
21          <title><?= $uc->title; ?></title>
22      </head>
23      <body>
24          <h1><?= $uc->title; ?></h1>
25
26          <?PHP if($uc->message) { ?>
27              <div><?= $uc->message; ?></div>
28          <?PHP } ?>
29
30          <main><?= $uc->content; ?></main>
31
32      </body>
33  </html>
```

Adding the UserController class

- Add a new file to the classes folder, call it **UserController.php.**
- This file will be an object which will house all of our application logic.
- This class requires a namespace of **classes** as it is to be found inside a folder of that name, this helps any code it contains locate itself relative to the project root.
- Add the code and comments below.

```php
<?PHP
    // namespace (folder path to class)
    namespace classes;

    // class declaration
    class UserController {

    }
?>
```

Member attributes

- This class will need a few attributes which should be added first.
- The **public** keyword denotes attributes which are in scope and to and usable by code outside of this class.
- The **private** keyword denotes attributes which are in scope only within this class.
- Add the highlighted code and comments.

```
5       // class declaration
6       class UserController {
7
8           // member variables
9           private $m_user;
10          private $m_dbConnector;
11
12          // public refs which get set to data required by the page
13          public $title;
14          public $content;
15          public $message = null;
16
17
18      }
```

Constructor function

- Constructor functions in PHP always follow the same syntax and are used to initialise data required when the class is instantiated .
- In this case the constructor checks the session for a logged in user and records the logged in user if found in the session, then calls a method of itself named **Router** (we will code this next).
- Add the highlighted code below.

```
15          public $message = null;
16
17          /**
18           * Constructor function
19           */
20          public function __construct() {
21              // grab user from the session if it exists - set null if not (ternary statement)
22              $this->m_user = (isset($_SESSION['user']))? $_SESSION['user'] : null;
23
24              // decide on what to do with the page request
25              $this->Router();
26          }
27      }
```

Router

- Many frameworks use a single landing page (**index.php**) and then employ a concept of a Router which makes the decisions about what action has been requested and what (if any) application logic needs to be processed to complete the request.
- **Laravel** and **CakePHP** use a system of **URL rewriting** and trapping to map routes to URLs (an example of **Laravel** routing is shown below).
- This project is not large enough to require something of that scale but we need some sense of routing and this should be an easily identified and isolated block of code, hence it gets its own function.

```
26  Auth::routes();
27  Route::get('/home', 'HomeController@index')->name('home');
28  Route::get('/home/admin', 'HomeController@admin')->name('admin');
```

- This code is an example, **DO NOT** add it to the project currently open

Adding a Router function

- Our router will make decisions about the presence of certain information contained in **$_POST**, **$_GET** and **$_SESSION.**
- The Router function will then call functions which will run application logic and will set the **$message**, **$title** and **$content** attributes ready for the final view to be drawn.
- We should take a moment to consider which functions will be needed.
- The application deals with user a user login system so it will need:
  - A login form for users who are not logged in but have an account
  - A login function to process a login request from the form
  - A register form for users who are not logged in and do not have an account
  - A register function to process a new account request from the form
  - A welcome page to greet the user when they successfully log in
  - A logout function for a user who has finished using the application
- We should get the login form and the register form views working first.

Login view

- Create a new file in the **views** folder with the filename **login.php.**
- This file contains a PHP string with a simple HTML login form inside.
- Note the **hidden** input called action with a value of **login** – this will be used by our router function.
- Also note the use of a hyperlink with a get value of register = 1 – this will also be used by our router function.
- Add the following code and save the view.

```
<?PHP
    return '

<p>Please use this form to login and begin, or <a href="index.php?register=1">register.</a></p>

<form action="index.php" method="post">
    <input type="hidden" id="action" name="action" value="login" />
    <input type="email" id="email" name="email" required placeholder="Enter your email address" />
    <input type="password" id="password" name="password" required placeholder="Enter your password"/>

    <input type="submit" value="Login"/>
</form>

'; ?>
```

Adding a Router function

- By default we should show the login form as this is the most likely thing that a user would want at the start of an interaction.
- For now we add a Router function to **UserController** which simply makes a call to another function called **setLoginPage.**
- Add the highlighted code and comments to **UserController.php.**

```
17          /**
18           * Constructor function
19           */
20          public function __construct() {
21              // grab user from the session if it exists - set null if not (ternary statement)
22              $this->m_user = (isset($_SESSION['user']))? $_SESSION['user'] : null;
23
24              // decide on what to do with the page request
25              $this->Router();
26          }
27
28          /**
29           * Router function (decide what do to based on type of query)
30           */
31          private function Router() {
32              $this->setLoginPage(); // show login page (default)
33          }
```

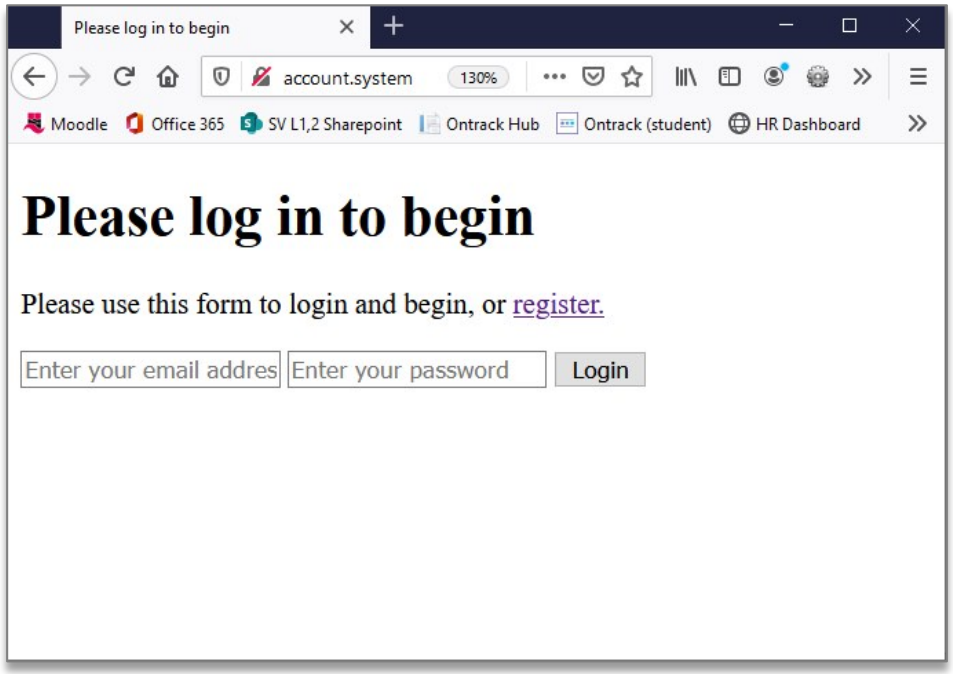
Adding setLoginPage

- Now that we have called **setLoginPage**, we should code it.
- This function simply assigns values to a few of the object attributes.
- It does nothing other than assign a view so no other application logic is present.
- Add the highlighted code to **UserController.php** and save all files.

```
28          /**
29           * Router function (decide what do to based on type of query)
30           */
31          private function Router() {
32              $this->setLoginPage(); // show login page (default)
33          }
34
35          /**
36           * Sets the view to represent the login page (default)
37           */
38          private function setLoginPage() {
39              $this->title = "Please log in to begin";
40              $this->content = require('views/login.php');
41          }
```

Testing the applicaiton

- The application does not do a lot at present, but should now be complete enough to serve our login form.
- Open a web browser and navigate to http://account.system.
- You should see a page similar to the one below.



Adding a register view

- Of course, nobody has an account yet as there is no user information in our database to login with.
- To fix this we can implement a register form.
- Create a new file in the **views** folder named **register.php.**
- Add the code below and save this file.

```
<?PHP
    // regrab any sent data just in case a mistake was made
    $email = (isset($_POST['email']))? $_POST['email'] : "";

    return '

<p>Please use this form to register a new account and begin, or <a href="index.php">login with an existing account.</a></p>

<form action="index.php" method="post">
    <input type="hidden" id="action" name="action" value="register" />
    <input type="email" id="email" name="email" required placeholder="Enter your email address" value="' . $email . '"/>
    <input type="password" id="password" name="password" required placeholder="Set a password"/>
    <input type="password" id="password_confirm" name="password_confirm" required placeholder="Re-enter the password"/>

    <input type="submit" value="Register"/>
</form>

'; ?>
```

Adding setRegisterPage

- Next, add a **setRegisterPage** function to **UserController.**
- This function also has no application logic and simply calls a view.
- Add the highlighted code to **UserController.php.**

```
35          /**
36           * Sets the view to represent the login page (default)
37           */
38          private function setLoginPage() {
39              $this->title = "Please log in to begin";
40              $this->content = require('views/login.php');
41          }
42
43          /**
44           * Sets the view to represent the register page
45           */
46          private function setRegisterPage() {
47              $this->title = "Please register to use the application";
48              $this->content = require('views/register.php');
49          }
```

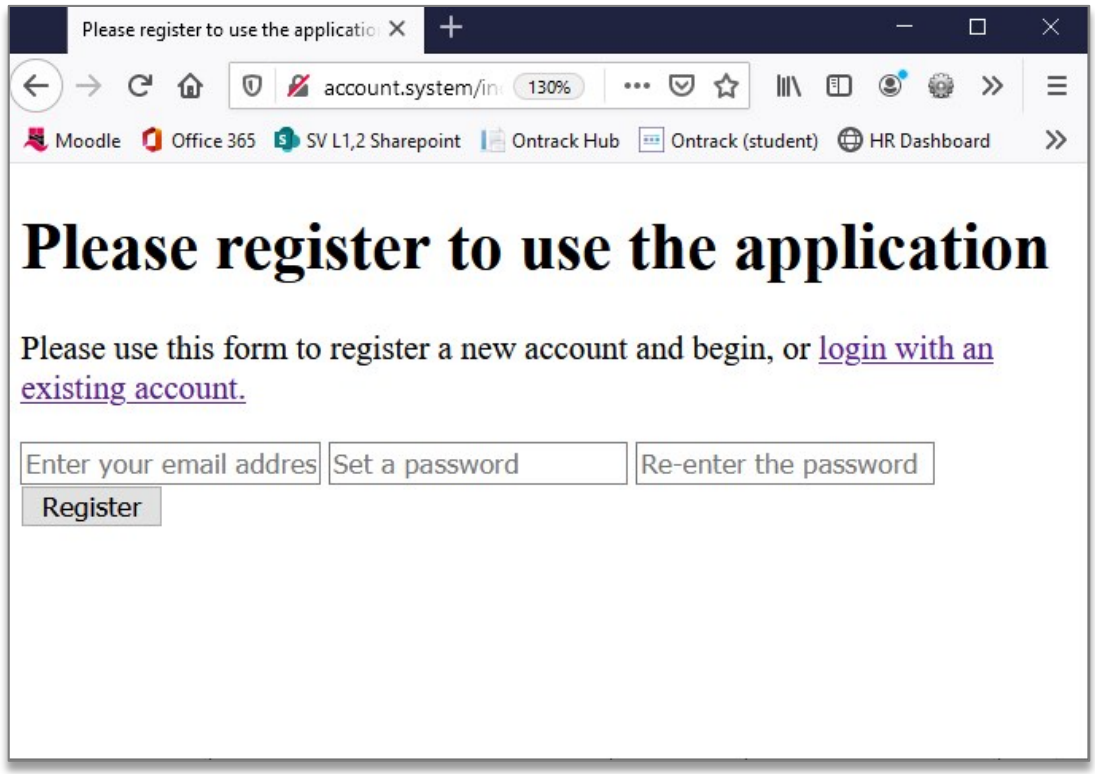
Routing to the registration page

- Amend the **Router** function to contain an **if / else** statement testing for the presence of **$_GET[‘register’].**
- We will add additional clauses to this function in coming stages, each to represent one of the functions identified for this application on slide 30.
- Change the Router function to add the highlighted code below.

```
28          /**
29           * Router function (decide what do to based on type of query)
30           */
31          private function Router() {
32              if (isset($_GET['register'])) $this->setRegisterPage(); // show registration page
33              else $this->setLoginPage(); // show login page (default)
34          }
```

Testing the application

- Refresh the page in the browser.
- You should now be able to navigate between login and register views using the hyperlinks at the top of each page.



DBConnector class

- In order for the system to work, it needs to connect to the database we created and setup earlier on.
- In order to make this easier, all of the DB connection logic, error trapping and so on will be wrapped in another class.
- This creates a neat blob of logic which is easily re-used in other projects as it will contain no functionality specific to this project beyond the connection account details.
- Create a new file in the **classes** folder with the name **DBConnector.php.**

Coding DBConnector

- This class will manage all of our database connection logic exposing methods to let us open, run specific queries and close a connection.
- Start the class with the code below which defines a new class in the classes namespace and adds attributes with database account information for this project.

```php
<?PHP
    // namespace
    namespace classes;

    // class declaration
    class DBConnector {

        // member variables
        private $m_username = "accountsystem";
        private $m_password = "accountsystem";
        private $m_dbname = "accountsystem";
        private $m_hostname = "localhost";
    }
?>
```

Coding DBConnector

- Two more member attributes are needed.
- The first will contain any error messages generated by the database and will provide visibility of these to the application.
- The second will be a reference to our **mysqli** connection object and will allow a number of methods to be run against the same connection without it being closed.*
- Add the highlighted code to **DBConnector.php.**

```
 8          // member variables
 9          private $m_username = "accountsystem";
10          private $m_password = "accountsystem";
11          private $m_dbname = "accountsystem";
12          private $m_hostname = "localhost";
13
14          private $m_errorText = null;
15          private $m_connectionref = null;
```

Coding DBConnector

- There is no initial setup needed and nothing to run by default in this class so no **constructor** method is needed.
- Instead we get on with coding a method which opens a database connection and stores a reference to it in the attribute we just added.
- Add the highlighted code to **DBConnector.php.**

```
14          private $m_errorText = null;
15          private $m_connectionref = null;
16
17          /**
18           * DBconnect function
19           *
20           * returns boolean connection state
21           */
22          public function Open() {
23              // open connection \ refers to global namespace needed to load mysqli
24              $this->m_connectionref = new \mysqli($this->m_hostname, $this->m_username, $this->m_password, $this->m_dbname);
25
26              // check connection, die if not present
27              if ($this->m_connectionref->connect_error) {
28                  $this->m_errorText = $this->m_connectionref->connect_error;
29                  return false;
30              }
31              return true;
32          }
```

Coding DBConnector

- Next add another method to allow the opposite action and close an active database connection.
- Add the highlighted code to **DBConnector.php.**

```
26              // check connection, die if not present
27              if ($this->m_connectionref->connect_error) {
28                  $this->m_errorText = $this->m_connectionref->connect_error;
29                  return false;
30              }
31              return true;
32          }
33
34          /**
35           * Close connection
36           */
37          public function Close() {
38              $this->m_connectionref->close();
39          }
```

Coding DBConnector

- Next add another method to run queries.
- This method is set as private for reasons which should become obvious in the next few steps.
- Add the highlighted code to **DBConnector.php.**

```
37        public function Close() {
38            $this->m_connectionref->close();
39        }
40
41        /**
42         * Query - this function actually does the work!
43         * private so only usable from inside this class
44         *
45         * returns boolean success indicator or results collection
46         */
47        private function Query($sql) {
48
49            // TODO - input sanitisation here!!!!!
50
51            // test query, set error text and fail if it does not complete
52            if (!$this->m_connectionref->query($sql)) {
53                $this->m_errorText = $this->m_connectionref->error;
54                return false;
55            }
56            return true;
57        }
```

Coding DBConnector

- Next add three more methods, each named for an SQL action, each calling the Query method that we just added.
- Each of these will run queries which do not return result collections.
- The first that we add is for the **UPDATE** sql action.
- Add the highlighted code to **DBConnector.php.**

```
47        private function Query($sql) {
48
49            // TODO - input sanitisation here!!!!!
50
51            // test query, set error text and fail if it does not complete
52            if (!$this->m_connectionref->query($sql)) {
53                $this->m_errorText = $this->m_connectionref->error;
54                return false;
55            }
56            return true;
57        }
58
59        /**
60         * Update function (wraps Query more explicity)
61         *
62         * returns boolean success indicator
63         */
64        public function Update($sql) {
65            return $this->Query($sql);
66        }
```

Coding DBConnector

- Add similar methods for **INSERT** and **DELETE.**
- Add the highlighted code to **DBConnector.php.**

```
64          public function Update($sql) {
65              return $this->Query($sql);
66          }
67
68          /**
69           * Insert function (wraps Query more explicity)
70           *
71           * returns boolean success indicator
72           */
73          public function Insert($sql) {
74              return $this->Query($sql);
75          }
76
77          /**
78           * Delete function (wraps Query more explicity)
79           *
80           * returns boolean success indicator
81           */
82          public function Delete($sql) {
83              return $this->Query($sql);
84          }
```

Coding DBConnector

- **SELECT** queries work a little differently as they return a result set if successful and nothing if not, so a different method is required.
- Add the highlighted code to **DBConnector.php.**

```
82          public function Delete($sql) {
83              return $this->Query($sql);
84          }
85
86          /**
87           * Select function (wraps Query more explicity)
88           *
89           * returns results collection
90           */
91          public function Select($sql) {
92
93              // TODO - input sanitisation here!!!!!
94
95              return $this->m_connectionref->query($sql);
96          }
```

Coding DBConnector

- Finally we add a final method to allow access to the contents of the **$m_errorText** attribute if needed.
- Add the highlighted code to the end of the class and save the file.

```
93          public function Select($sql) {
94
95              // TODO - input sanitisation here!!!!!
96
97              return $this->m_connectionref->query($sql);
98          }
99
100         /**
101          * ErrorMessage - getter for any error text generated
102          *
103          * returns string contained in $this->m_errorText
104          */
105         public function ErrorMessage() {
106             return $this->m_errorText;
107         }
108
109     }
110 ?>
```

Using DBConnector

- We now have a means with which to connect to our database.
- We can enable **UserController** to use the **DBConnector** class by adding a use statement for it and by setting the reference for it in the **constructor.**
- Add the highlighted code to **UserController.php.**

```
2       // namespace (folder path to class)
3       namespace classes;
4
5       // load the db connector
6       use classes\DBConnector;
7
8       // class declaration
9       class UserController {
10
11          [...]
12
13          /**
14           * Constructor function
15           */
16          public function __construct() {
17              // grab user from the session if it exists - set null if not (ternary statement)
18              $this->m_user = (isset($_SESSION['user']))? $_SESSION['user'] : null;
19
20              // create a new DB connector
21              $this->m_dbConnector = new DBConnector();
22
23              // decide on what to do with the page request
24              $this->Router();
25          }
```

Registering a new user

- If a new user wishes to create an account and fills in the register form, we can contact the database and create an account for them.
- The first thing our **doRegister** function should do is check that both passwords submitted match and to generate an appropriate error if not.
- Add the highlighted code on the next slide to **UserController.php.**

Registering a new user

```
50          /**
51           * Sets the view to represent the register page
52           */
53          private function setRegisterPage() {
54              $this->title = "Please register to use the application";
55              $this->content = require('views/register.php');
56          }
57
58          /**
59           * Sets the view to represent the register page
60           */
61          private function doRegister() {
62              // check for password match
63              if ($_POST['password'] !== $_POST['password_confirm']) {
64                  // passwords do not match - error and exit
65                  $this->message = "Passwords do not match!";
66                  $this->title = "Please register to use the application";
67                  $this->content = require('views/register.php');
68                  return;
69              }
70
71              // TODO - save record
72
73          }
```

Registering a new user

```
50          /**
51           * Sets the view to represent the register page
52           */
53          private function setRegisterPage() {
54              $this->title = "Please register to use the application";
55              $this->content = require('views/register.php');
56          }
57
58          /**
59           * Sets the view to represent the register page
60           */
61          private function doRegister() {
62              // check for password match
63              if ($_POST['password'] !== $_POST['password_confirm']) {
64                  // passwords do not match - error and exit
65                  $this->message = "Passwords do not match!";
66                  $this->title = "Please register to use the application";
67                  $this->content = require('views/register.php');
68                  return;
69              }
70
71              // TODO - save record
72
73          }
```

Registering a new user

- Next, we know that passwords match so we should create a new account for this user.
- First we open a connection to the database using the **Open** method of the **DBConnector** class created earlier.
- Second, we compose an SQL **INSERT** statement to add this new account.
- Add the highlighted code to the **doRegister** function.

```
61        private function doRegister() {
62            // check for password match
63            if ($_POST['password'] !== $_POST['password_confirm']) {
64                // passwords do not match - error and exit
65                $this->message = "Passwords do not match!";
66                $this->title = "Please register to use the application";
67                $this->content = require('views/register.php');
68                return;
69            }
70
71            // all ok - make db connection
72            $this->m_dbConnector->Open();
73
74            // build SQL
75            $sql = "INSERT into users (email,password) VALUES ('" . $_POST['email'] . "','" . $_POST['password'] . "')";
```

Registering a new user

- We are ready to run our SQL statement using the **Insert** method of the **DBConnector** class and examining the response for errors.
- If **true**, the insert succeeded and we can log the user in.
- If **false**, the insert failed and we should display an error message and return to the register account form.
- Finally, in any event, it is good practice to close off any outstanding database connections which we can do using the **Close** method of the **DBConnector** class created earlier.
- The code to accomplish all of this is on the next slide.

Registering a new user

- Add the highlighted code to the end of the **doRegister** function.

```
74              // build SQL
75              $sql = "INSERT into users (email,password) VALUES ('" . $_POST['email'] . "','" . $_POST['password'] . "')";
76
77              // run query, check for errors
78              if ($this->m_dbConnector->Insert($sql)) {
79
80                  // all okay, set the user login in the session
81                  $_SESSION['user'] = $_POST['email'];
82
83                  // show welcome
84                  $this->message = "New account registered for " . $_POST['email'];
85                  $this->title = "Welcome to the application";
86                  $this->content = require('views/welcome.php');
87              }
88              else {
89                  // error found
90                  $this->message = $this->m_dbConnector->ErrorMessage();
91                  $this->title = "Please register to use the application";
92                  $this->content = require('views/register.php');
93              }
94
95              // close db connection
96              $this->m_dbConnector->Close();
97          }
```

Adding the routing for doRegister

- We need a new clause in the **Router** method of **UserController.**
- In this instance a form submission has occurred which means that data will have been sent to the application using **post** which is the method set in the HTML registration form.
- This form contains a hidden field called **action** with its value set to **register** so this is what we check for.
- Add the highlighted code to the Router function.

```
34        /**
35         * Router function (decide what do to based on type of query)
36         */
37        private function Router() {
38            if (isset($_GET['register'])) $this->setRegisterPage(); // show registration page
39            else if (isset($_POST['action']) && $_POST['action'] === "register") $this->doRegister(); // do registration
40            else $this->setLoginPage(); // show login page (default)
41        }
```

Welcome page view

- As we specified a view called **welcome.php**, we should add it.
- This page will be displayed whenever a user is successfully logged in.
- The only control we need to add here is a **logout** control.
- Create a new file in the views folder with the filename **welcome.php.**
- Add the following code and save the file.

```
1  <?PHP
2      return '
3      <p>Welcome to the application</p>
4
5      <p>You have been successfully logged in as ' . $_SESSION['user'] . '.</p>
6
7      <p><a href="index.php?logout=1">Logout</a></p>
8
9      ';
10 ?>
```

More routes

- We now have accounts that can be logged in so the router needs to trap any request from a user who is already logged in.
- We also should add the ability to reverse this with a logout function.
- The first check we should make is for a logout request as this supersedes all other requests.*
- Next we check if the user is already logged in and load the welcome view if we find that they are.
- Add the highlighted code to the **Router** function in **UserController.**

```
34        /**
35         * Router function (decide what do to based on type of query)
36         */
37        private function Router() {
38            if (isset($_GET['logout'])) $this->doLogOut(); // do logout, then show login page
39            else if ($this->m_user) $this->setWelcomePage(); // already logged in, show welcome page
40            else if (isset($_GET['register'])) $this->setRegisterPage(); // show registration page
41            else if (isset($_POST['action']) && $_POST['action'] === "register") $this->doRegister(); // do registration
42            else $this->setLoginPage(); // show login page (default)
43        }
```

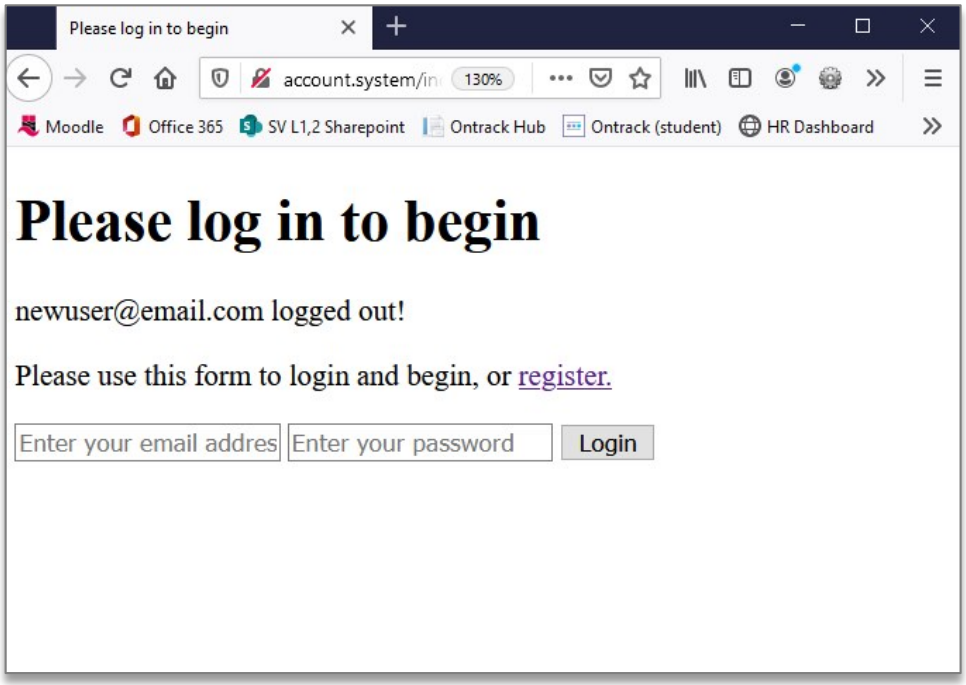
Adding doLogout

- The Router now expects a function named **doLogout.**
- This is a simple function which clears any user data from the session and then returns the user to the login form.
- Add the highlighted code to the bottom of the **UserController** class.

```
93                  $this->message = $this->m_dbConnector->ErrorMessage();
94                  $this->title = "Please register to use the application";
95                  $this->content = require('views/register.php');
96              }
97
98              // close db connection
99              $this->m_dbConnector->Close();
100         }
101
102         /**
103          * Process a logout
104          */
105         private function doLogOut() {
106             // set a status message
107             $this->message = $this->m_user . " logged out!";
108
109             // clear logged in information
110             $_SESSION['user'] = null;
111             $this->m_user = null;
112
113             // get login form
114             $this->setLoginPage();
115         }
116     }
117 ?>
```

Testing registration and logout

- Reload the application in your browser.
- Test the registration form, create a new user who should be automatically logged in (we still cannot use the login form at this point).
- Test the logout control.



Login function

- The last element to include is the application logic behind a login request for an existing user.
- First we need to add a route for a detected submission from the login form which inspects the **$_POST['action']** field similarly to **register.**
- Add the highlighted code to the **Router** function in **UserController** to complete the **Router** function.*

```
34        /**
35         * Router function (decide what do to based on type of query)
36         */
37        private function Router() {
38            if (isset($_GET['logout'])) $this->doLogOut(); // do logout, then show login page
39            else if ($this->m_user) $this->setWelcomePage(); // already logged in, show welcome page
40            else if (isset($_GET['register'])) $this->setRegisterPage(); // show registration page
41            else if (isset($_POST['action']) && $_POST['action'] === "register") $this->doRegister(); // do registration
42            else if (isset($_POST['action']) && $_POST['action'] === "login") $this->doLogin(); // do login
43            else $this->setLoginPage(); // show login page (default)
44        }
```

Adding setWelcomePage

- The already logged in route specifies a function called **setWelcomePage.**
- This is another small function which simply returns a view similar to **setRegisterPage** and **setLoginPage.***
- Add the highlighted code to implement this function.

```
54          /**
55           * Sets the view to represent the register page
56           */
57          private function setRegisterPage() {
58              $this->title = "Please register to use the application";
59              $this->content = require('views/register.php');
60          }
61
62          /**
63           * Sets the welcome page to represent a logged in user
64           */
65          private function setWelcomePage() {
66              $this->title = "Welcome to the application " . $this->m_user;
67              $this->content = require('views/welcome.php');
68          }
```

Adding the doLogin function

- Start by building an SQL statement to retrieve the user record from the database using both their **username** and **password** (if a record is returned, they have the right details and should be allowed to log in).
- Next open a connection to the database using the **DBConnector** object.
- Add the highlighted code to the end of the **UserController** class.

```
110             // clear logged in information
111             $_SESSION['user'] = null;
112             $this->m_user = null;
113
114             // get login form
115             $this->setLoginPage();
116         }
117
118         /**
119          * Sets the view to represent the register page
120          */
121         private function doLogin() {
122
123             // build sql
124             $sql = "SELECT email,password from users WHERE email='" . $_POST['email'] . "' and password='" .
                $_POST['password'] . "' Limit 1 ";
125
126             // open connection to db
127             $this->m_dbConnector->Open();
128
129         }
```

Adding the doLogin function

- Run the SQL query using the Select method of DBConnector.
- Store the response in **$result.**
- Check how many rows (records) are returned (should be either 0 or 1).
- If no records are returned, the login has failed, set **$message**, call the login form view with **doLoginForm** and exit the function with **return.**
- Whether successful or not, close the DB connection.
- The code for this stage is on the following slide.

Adding the doLogin function

- Add the highlighted code to the **doLogin** function block.

```
118         /**
119          * Sets the view to represent the register page
120          */
121         private function doLogin() {
122 
123             // build sql
124             $sql = "SELECT email,password from users WHERE email='" . $_POST['email'] . "' and password='" .
                $_POST['password'] . "' Limit 1 ";
125 
126             // open connection to db
127             $this->m_dbConnector->Open();
128 
129             // run query, get user result
130             $result = $this->m_dbConnector->Select($sql);
131 
132             // close connection
133             $this->m_dbConnector->Close();
134 
135             // check for user, if none found - error and exit
136             if ($result->num_rows < 1) {
137                 $this->message = "The email or password are incorrect!";
138                 $this->setLoginPage();
139                 return;
140             }
141         }
```

Completing the doLogin function

- Finally, we deal with a successful login.
- We convert the returned record to an associative array.
- We take the email from the returned record (safer) and add it into **$_SESSION['user']** as well as set it in **$this->m_user** to indicate that this user is logged in.
- Finally we draw the welcome view with **setWelcomePage** and set the successful login message in **$message.**
- The final block of code to add is on the following slide.

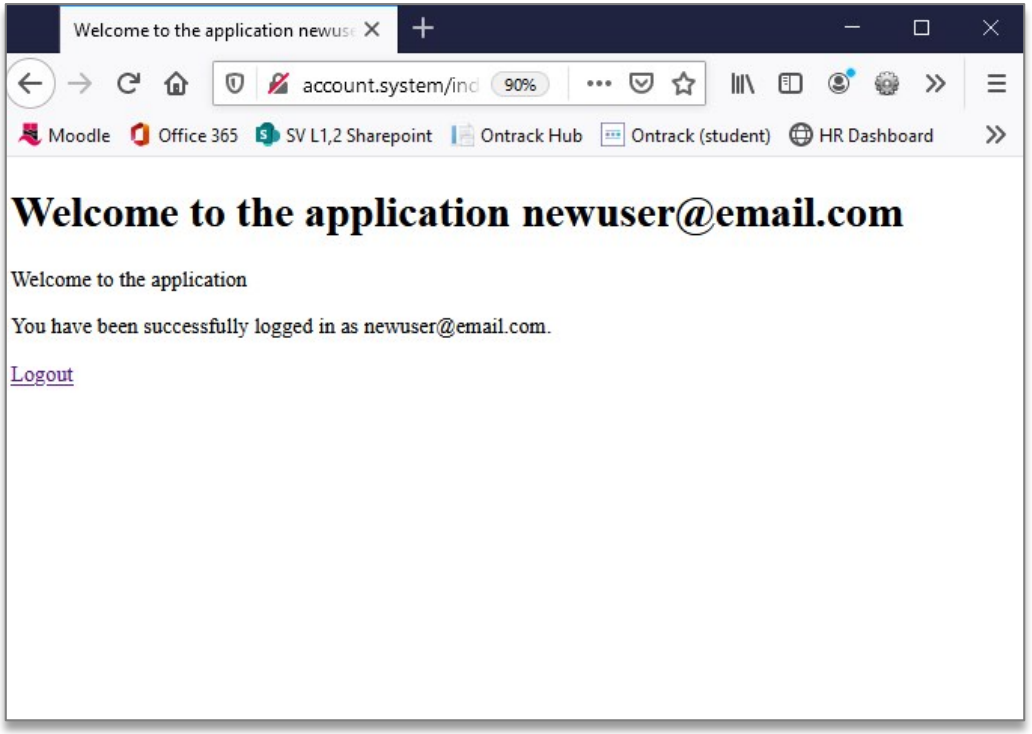
Completing the doLogin function

- Add the highlighted code to the bottom of the **doLogin** function.

```
143             // check for user, if none found - error and exit
144             if ($result->num_rows < 1) {
145                 $this->message = "The email or password are incorrect!";
146                 $this->setLoginPage();
147                 return;
148             }
149
150             // get dictionary object for user
151             $user = $result->fetch_assoc();
152
153             // set session and logged in user
154             $_SESSION['user'] = $user['email'];
155             $this->m_user = $user['email'];
156
157             // do logged in view
158             $this->message = "Logged in " . $user['email'];
159             $this->setWelcomePage();
160         }
161     }
162 ?>
```

Final tests

- Reload the browser and try to log in with an existing user account.
- Remember to test both correct and incorrect data to make sure that incorrect account details are rejected.



Extensions

- The issue with this project is that we save user passwords into our database in plain text – this is an obvious security flaw.
- Research one-way hash encryption techniques used in PHP and convert the application to store only encrypted passwords.
- **Note:** two-way encryption techniques are also possible in PHP but if the string can be unencrypted it is less secure and should not be used for known data such as passwords.

Project review

- Web Services serve database information through an API.
- This API is often driven by a server-side framework (Laravel, etc).
- PHP frameworks commonly employ an OOP approach which is structured using an MVC design pattern or something very close to it.
- This account system project uses something akin to an MVC and OOP approach whilst strengthening understanding of previous topics such as connecting to a MySQL database using PHP.
- We started this unit with a gentle introduction to server-side coding
- We have built experience, knowledge and understanding to the point that we now know enough to start using a server-side framework.
- Next week we will install and configure Laravel.
- Please make sure that you have Composer installed in readiness.

BSc Applied Computing

Session review

Recap

- Recap PHP and MySQL connectivity
  - Last exercise progress
  - Reading list
  - Stateful vs RESTful
  - Use of the PHP session
- PHP and Object-orientation
  - PHP and OOP
  - Namespace and foldering
  - Using a specific class
  - MVC and frameworks
- Simple OOP user account system
  - Project setup
  - Project development workshop

Any questions?

- Next…
- PHP frameworks, an introduction to Laravel