WEB 306 – PHP: Databases and Framework

Day 7 – MVC and CRUD

1. CRUD

1.1 The CRUD Cycle

The average web application should be able to perform four basic actions with data: ***Create***, ***Read***, ***Update*** and ***Delete***. These actions are represented by the acronym ***CRUD*** and make up what is known as ***The CRUD Cycle***.

> "The CRUD cycle describes the elemental functions of a persistent database. CRUD stands for Create, Read, Update and Delete. (Retrieve may occasionally be substituted for Read.) These functions are also descriptive of the data life cycle."
>
> — From CRUD cycle (Create, Read, Update and Delete Cycle), TechTarget

1.2 CRUD Applications

Examples of CRUD applications include web applications:

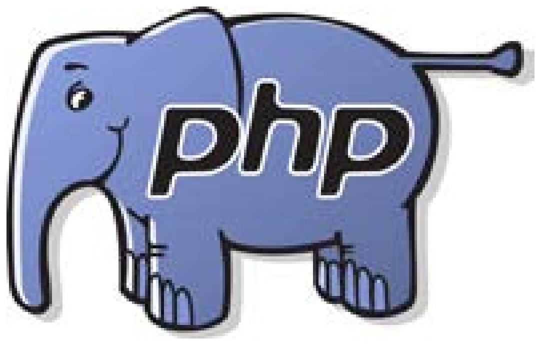
1. **GMail, Outlook, etc.**
2. **Google Drive, OneDrive, Dropbox etc.**
3. **GitHub**

A CMS or Content Management System would also be a CRUD application:

1. **Wordpress**
2. **Drupal**
3. **Joomla**
4. **ExpressionEngine**

With the exception of Twitter, most social media networks would also be considered CRUD applications:

1. **Facebook**
2. **Instagram**
3. **Medium**
4. **Tumblr**



PHP Resources

1. PHP Website
2. XAMPP
3. Documentation
4. W3Schools
5. CSS Tricks
6. Scotch.io
7. Stack Overflow

2. MVC

2.1 What is MVC?

MVC stands for ***Model-View-Controller***. It is an architechtural pattern for developing applications which separates a program into three distinct but interconnected parts.

It was designed by a Norweigian computer scientist named Trygve Reenskaug, in 1979 for the Smalltalk language, in order to streamline GUI (Graphic User Interface) programs which allowed users to control large amounts of data.

> **MVC is an application design model comprised of three interconnected parts. They include the model (data), the view (user interface), and the controller (processes that handle input).**
>
> **— From MVC, TechTerms**

MySQL Resources

1. MySQL Website
2. MySQL Documentation
3. XAMPP
4. W3Schools
5. Stack Overflow

MariaDB Resources

1. MariaDB Website
2. MariaDB Documentation
3. XAMPP
4. Stack Overflow

2.2 The Model

The Model is a representation of whichever data the program is using. This could be connected to a database or it could simply be a file or an object. This is where the data is ***modelled***. In the context of a PHP application, examples of our Model would be our Car object data or our SQL database.

2.3 The View

The View is the front-end of the application, or whatever elements the user can see or interact with through the user-interface. This is where the data is ***viewed***. In the context of a PHP application, our View would be our HTML templating files and would also include assets such as CSS, Sass and JavaScript (although in the case of JavaScript, you could also have a full MVC framework on the front-end such as ***Angular*** or ***EmberJS***, but this approach has become less popular for front-end development in recent years).

2.4 The Controller

The Controller controls the flow of data from the Model to the View as well as inversely from the View to the Model. This is where the data is ***controlled***. In the context of a PHP application, our Controller would be the code we used to collect data from submitted forms and to send that data to either the database (the Model) or the HTML page (the View).

2.5 MVC Frameworks

Web frameworks provide a pre-planned archtechtural pattern for designing applications using a given language. Most modern web frameworks follow MVC principles as well as OOP and CRUD principles. Examples of MVC frameworks include:

1. **Laravel (PHP)**
2. **Zend Framework (PHP)**
3. **CakePHP (PHP)**
4. **Phalcon (PHP)**
5. **Django (Python)**
6. **TurboGears (Python)**
7. **Ruby on Rails (Ruby)**
8. **Angular (JavaScript)**
9. **EmberJS (JavaScript)**
10. **SailsJS (Node.js)**

PHPMyAdmin Resources

1. **PHPMyAdmin Website**
2. **XAMPP**
3. **PHPMyAdmin Documentation**
4. **PHPMyAdmin Support**
5. **PHPMyAdmin Tutorials**
6. **Stack Overflow**

3. Utility Classes

SQL Resources

1. **XAMPP**
2. **W3Schools**
3. **Scotch.io**
4. **Stack Overflow**

3.1 Static Classes

Utility classes, or helper classes, are classes which are used as libraries for grouping related methods and properties together to be used by other classes, or sometimes outside of classes, and are not used to model data independently. They are normally classes which contain ***only static methods and properties***.

Static classes would be used when you have a class which is used fairly commonly accross your program by other classes or even outside of classes but is not reliant on an instance of a class. In general, you would want to use static classes relatively sparingly with the exception of utility classes. Some PHP developers consider it a bad practice to use static methods and properties altogether, but this is a subjective opinion. The reality is that the use of static properties and methods is fairly common, to the point where they are heavily utilized by popular MVC frameworks such as Laravel.

3.2 The Sanitizer Class

Our user data must be sanitized and validated but we should not define all of these methods in our Model class like we did when we used them with our `Car` class. According to the ***Single Responsibility Principle*** of object-oriented programming, a class should only have a single reason to change. Therefore, to keep things more organized and to keep our responsibilites separated, we will create a dedicated utility class for defining sanitization and validation methods.

```
class Sanitizer {
    private static $chars = ‘/([^a-zA-Z\-. ]+)/’;
    private static $name_chars = ‘/([^a-zA-Z0-9\-_.]+)/’;

    public static function filter_input($string, $chars = ‘//’) {
         $trim_space = trim($string);
         $strip_tags = strip_tags($trim_space);
         $remove_chars = preg_replace($chars, ‘’, $strip_tags);
         return $remove_chars;
    }

    public static function format_name($name) {
         $filtered = self::filter_input($name, self::$name_chars);
         $lowercase = strtolower($filtered);
         $capitalized = ucwords($lowercase);
    }

    public static function filter_file($input_file, $files_array) {
         $filtered = self::filter_input($input_file, self::$chars);
         $file_name = strtolower($filtered);

         foreach ($files_array as $item_name) {
              if ($item_name == $file_name) {
                   return $file_name;
              } // end if
         } // end foreach
    } // end filter_file

    public static function filter_number($number) {
         $filter_int = filter_var($number, FILTER_SANITIZE_NUMBER_INT);
         return (int) $filter_int;
    }

    public static function escape_html($string) {
         $convert = mb_convert_encoding($string, ‘UTF-8’, ‘UTF-8’);
         $encode_html = htmlspecialchars($string, ENT_QUOTES, ‘UTF-8’);
         return $encode_html;
    }
```

3.3 Singleton Classes

A singleton class is another type of utility class which is similar to a static class in that you do not need to create a new instance of the class every time you use it. However, a singleton class does require an instance of a class. What makes a singleton class different is that a new instance of a class is defined with a private constructor method and is only defined once, inside of the class. This instance is then stored in a static property and static getter methods are used to access it.

3.4 The Connector Class

The advantage of using singletons is that an object which needs to be referenced many times only needs to be instantiated once, which improves performance. Singletons are frequently used to connect to databases to avoid creating multiple database connections unnecessarily. We will create a singleton class to connect to our database using PDO.

Our application will use a database table with two columns. The first will be named "id" and will be an **INT** which will be set to `AUTO_INCREMENT` and will be defined as our `PRIMARY KEY`. The second column will hold serialized, encoded PHP objects and will be a BLOB which is named "object".

id 🔑	object
1	Tzo4OiJFbGVwaGFudCI6NDp7czo3OiIA
2	Tzo4OiJFbGVwaGFudCI6NDp7czo3OiIA
3	Tzo4OiJFbGVwaGFudCI6NDp7czo3OiIA

```
class Connector {
    private static $instance = null;
    private $host;
    private $name;
    private $user;
    private $password;

    private function __construct() {
        $this->host = "localhost";
        $this->name = "elephants";
        $this->user = "root";
        $this->password = "";

        $pdo = new PDO(
            'mysql:host=' . $this->$host .
            ';dbname=' . $this->$name,
            $this->$user,
            $this->$password
        );
    }

    public static function get_instance() {
        if (self::$instance == null) {
            self::$instance = new Connector;
        }
        return self::$instance;
    }

    public static function get_connection() {
        return $this->pdo;
    }
}
```

4. Creating an MVC CRUD Application

4.1 The Elephant Model

We are going to make a CRUD application using an object-oriented approach and following the MVC pattern. The first thing we need to do is create a class for our Model which will represent the data we are using in our app. We will be making an elephant database so we will make an `Elephant` class. We will want to do sanitization for our properties so we will require the `Sanitizer` class we made eariler. In order to keep our properties encapsulated we will want to use private or protected properties with getters and setters.

```
require '../utilities/Sanitizer.php';

class Elephant {
   protected $name;
   protected $age;
   protected $image;
   protected $image_files = [
         'bananas.jpg',
         'beach.jpg',
         'big-ball.jpg',
         'birds.jpg'
   ];

   public function __construct(array $input_values) {
         $this->set_name($input_values['name']);
         $this->set_age($input_values['age']);
         $this->set_image($input_values['image']);
   }
```

The `__get()` magic method is used as a universal getter method which gets the value of any private or protected property. Like the `__construct()` method it is not necessary to call the `__get()` method so you can use private and protected properties as if they are public.

There is also a `__set()` magic method but we won't use that as we want to do unique sanitization and validation for each of our properties using setter methods.

```
   public function __get($property) {
         if (property_exists($this, $property)) {
              return $this->$property;
         }
   }

   public function set_name(string $value) {
         $this->name = Sanitizer::format_name($value);
   }

   public function set_age($value) {
         $this->top_speed = Sanitizer::filter_number($value);
   }

   public function set_image(string $value) {
         $this->image = Sanitizer::filter_file($value,
                $this->image_files);

            if (empty($this->image)) {
                $this->image = $this->image_files[0];
            }
        }
    }
```

4.2 The Elephant Controller

4.2.1 Required Classes

We will also need a Controller for our `Elephant` Model which will control the flow of data from the Model to the View and from the View to the Model. Our controller will manage our CRUD operations using four methods called `create()`, `read()`, `update()` and `delete()`. We will need to require both our database `Connector` class and our `Elephant` model.

```
    require '../utilities/Connector.php';
    require '../models/Elephant.php';
```

4.2.2 The Create Method

Our `create()` method will need a parameter to take in the user input. The parameter will be a a placeholder for the `$_POST` array.

```
    class ElephantController {
        public function create(array $post) {
```

We will check to see if our required fields have been filled out.

```
            if (!empty($post['name']) && !empty($post['age']) &&
                !empty($post['image'])) {
```

We will create a new instance of our `Elephant` class and its constructor method will take the `$_POST` placeholder.

```
                $elephant = new Elephant($post);
```

Then we will serialize the `Elephant` object so that we can store it in one column.

```
                $elephant = serialize($elephant);
```

When storing serialized objects in a database it is best practice to use the `base64_encode()` function to encode the serialized string so that the data does not get corrupted while it is being transferred and for added security.

```
                $elephant = base64_encode($elephant);
```

We then need to use our `Connector` class' instance getter to get the instance. We will then use our `get_pdo()` getter to get the PDO object and store it in a variable. We can then use an `INSERT INTO` statement and PDO's `prepare()` method to prepare our SQL query to insert the encoded car into the `object` column.

```
                $db = Connector::get_instance();
                $pdo = $db->get_pdo();
                $sql = "INSERT INTO `elephants` (`object`) VALUES
                     ('${elephant}')";
                $query = $pdo->prepare($sql);
```

We will use PDO's execute() method to execute the query inside an if statement to check to see if the query runs successfully. If so redirect to the "view" page with a success message, else go back to the "add" page with an error message.

```
                if ($query->execute()) {
                     header('Location: ../views/view.php?add=success');
                     exit();
                } else {
                     header('Location: ../views/add.php?db=error');
                     exit();
                }
```

If the required fields have been filled out, run the code, else go back to the "add" page with an error message.

```
          } else {
               header('Location: ../views/add.php?add=error');
               exit();
          }
     }
```

4.2.3 The Read Method

The read() method will be used to retreive data from the database. We then need to use our Connector class' instance getter to get the instance. We will then use our get_pdo() getter to get the PDO object and store it in a variable. We can then use a SELECT * FROM statement and PDO's prepare() method to prepare our SQL query to select all rows from the database.

Then we need to use PDO's execute() method to run the query. To store the database rows as an array, we need to use PDO's fetchAll() method and save it as a variable

```
     public function read() {
          $db = Connector::get_instance();
          $pdo = $db->get_pdo();
          $sql = "SELECT * FROM `elephants`";
          $query = $pdo->prepare($sql);
          $query->execute();
          $elephants = $query->fetchAll();
```

Because we used the base64_encode() function to encode our serialized object we must use the base64_decode() function to decode it. Because we used the serialize() function to convert our object data to a string we must use the unserialize() function to convert the data back to its initial format. We will loop through the elephants, decode and unserialized each one and then redefine them. We will then return the elephants as an array.

```
          foreach ($elephants as $elephant_num => $elephant_row) {
               $an_elephant = base64_decode($elephant_row);
               $an_elephant = unserialize($an_elephant);
```

```
                $elephants[$elephant_num]['object'] = $an_elephant;
            }
            return $elephants;
        }
```

4.2.4 The Update Method

The `update()` method is very similar to the create method with a couple of differences. We need to collect the ID of the elephant row from the `$_POST` array to identify which elephant we are updating.

```
   public function update(array $post) {
            $id = $post['id'];
```

Even though we're editing existing data we still need to check to make sure that the user did not leave any text fields blank.

```
            if (!empty($post['name']) && !empty($post['age']) &&
                !empty($post['image'])) {
```

We will create a new instance of our `Elephant` class to overwrite the old one and then serialize and encode it again.

```
                $elephant = new Elephant($post);
                $elephant = serialize($elephant);
                $elephant = base64_encode($elephant);
```

We can then use an `UPDATE` statement and PDO's `prepare()` method to write an SQL query to update `elephants` and set `object` to `$elephant` where the `id` matches `$id`.

```
                $db = Connector::get_instance();
                $pdo = $db->get_pdo();
                $sql = "UPDATE `elephants` SET object='${elephant}' WHERE
                     id=${id}";
                $query = $pdo->prepare($sql);
```

As before, we will use PDO's `execute()` method to execute the query inside an if statement to check to see if the query runs successfully. If so redirect to the "view" page with a success message, else go back to the "edit" page with an error message. However, when we redirect back to the "edit" page it will be necessary to pass the `$id` variable back to the page to identify the elephant that is being edited.

```
                if ($query->execute()) {
                     header('Location: ../views/view.php?add=success');
                     exit();
                } else {
                     header('Location: ../views/edit.php?db=error&id=' .
                          $id);
                     exit();
                }
            } else {
                header('Location: ../views/edit.php?edit=error&id=' .
                     $id);
                exit();
            }
```

```
    }
```

4.2.5 The Delete Method

The `delete()` method will be very similar to the update method. We still need to collect the ID of the elephant row from the `$_POST` array to identify which elephant we are deleting. However, We don't need to check if the user has filled out any fields or create a new instance of the `Elephant` class as we are simply deleting data.

```
    public function delete(array $post) {
        $id = $post['id'];
```

We can then use an `DELETE FROM` statement and PDO's `prepare()` method to write an SQL query to delete from `elephants` where the `id` matches `$id`.

```
        $db = Connector::get_instance();
        $pdo = $db->get_pdo();
        $sql = "DELETE FROM `elephants` WHERE id=${id}";
        $query = $pdo->prepare($sql);
```

As before, we will use PDO's `execute()` method to execute the query inside an if statement to check to see if the query runs successfully. If so redirect to the "view" page with a success message, else go back to the "view" page with an error message.

```
        if ($query->execute()) {
            header('Location: ../views/view.php?add=success');
            exit();
        } else {
            header('Location: ../views/view.php?db=error');
            exit();
        }
    }
  }
```

4.2.6 Controller Routes

None of our methods will run without being called. We will call the create, update and delete methods by pointing the action of a form to an address with a `$_GET` key of "action" and a value which matches the name of the method being called.

First we need to require our `ElephantController` class and create a new instance of our Controller. Then we will use a switch statement to check the value fo the "action" key and call the corresponding method in reponse.

```
  require 'ElephantController.php';

  $controller = new ElephantController;

  $action = $_GET['action'];

  switch ($action) {
    case 'create':
        $controller->create($_POST);
```

```
          break;
     case 'update':
          $controller->update($_POST);
          break;
     case 'delete':
          $controller->delete($_POST);
          break;
     default:
          header('Location: ../views/view.php');
          exit();
  }
```

If the key does not match any of the methods then the file will just redirect back to the "view" page.

4.3 The Views

4.3.1 The Add Elephant Page

The View file for the Add Elephant page will have a form on it with an action attribute which points to the address ***"../controllers/routes.php?action=create"*** in order to call the `create()` method in our Routes file. The name attribute of each input or select field must match with the key we used to retreive the data from the `$_POST` array in our model. We also must have a button tag with a type of "submit".

```
  <form action="../controllers/routes.php?action=create" method="post">
     <label for="name">Name:</label>
     <input type="text" name="name" value="" id="name">

     <label for="age">Age:</label>
     <input type="number" name="age" value="" id="age">

     <select name=”image” id=”image”>
          <option value=”bananas.jpg">Bananas</option>
          <option value=”beach.jpg">Beach</option>
          <option value=”big-ball.jpg">Big Ball</option>
          <option value=”birds.jpg">Birds</option>
          <option value=”dog.jpg">Dog</option>
     </select>

     <button type="submit"><span class="fa fa-plus"></span>
          Add Elephant</button>
  </form>
```

4.3.2 The View Elephants Page

The View file for the View Elephants page will display all of the elephants. In order to do so we must require our `ElephantController` class and create a new instance of our Controller. We can then call our `read()` method to get an array of all of the elephant rows.

```
  <div class="grid">
```

```
<?php
require '../controllers/ElephantController.php';

$controller = new ElephantController;
$elephants = $controller->read();
```

We can then loop through the array of elephants using a foreach loop and use the "object" key to get the object. We can also use the "id" column to get the elephant's unique ID.

```
foreach ($elephants as $elephant_num => $elephant_row):
   $elephant = $elephant_row['object'];
   $id = $elephant_row['id'];
   ?>
   <div class="elephant">
        <h2><?php echo Sanitizer::escape_html($elephant->name); ?></
h2>
        <h3><?php echo Sanitizer::escape_html($elephant->age); ?></h3>

        <p><img src="/views/img/<?php echo
             Sanitizer::escape_html($elephant->image); ?>"
             alt="<?php echo Sanitizer::escape_html($elephant->name);
             ?>"></p>
```

On the View Elephants page, inside of the loop, we will have an Edit button and a Delete button associated with each elephant.

In order for the delete button to work we need to create a form. The form will point to the address ***"../controllers/routes.php?action=delete"*** in order to call the `delete()` method in our Routes file. This form needs to have a hidden input tag which will have a name of "id". The value will echo the ID of the current elephant into the value attribute to pass the ID to the Controller. The "submit" button of the form will act as the Delete button.

To create the Edit button we can place a simple anchor tag in the form which links to the Edit Elephant page with a `$_GET` key of "id" and a value which will echo the ID of the current elephant into to pass the ID to the Controller

```
        <form action="../controllers/routes.php?action=delete"
             method="post">

             <a href="edit.php?id=<?php echo $id; ?>"
             class="button"><span class="fa fa-edit"></span>Edit</a>

             <input type="hidden" name="id" value="<?php echo $id; ?>"
                  id="id">

             <button type="submit"><span class="fa fa-trash"></span>
                  Delete</button>
        </form>
   </div>
   <?php
endforeach;
?>
```

```
</div>
```

4.3.3 The Edit Elephant Page

Finally to create the View file for the Edit Elephant page we must collect the "id" from the `$_GET` variable. Then we must require our `ElephantController` class and create a new instance of our Controller. We can then call our `read()` method to get an array of all of the elephant rows.

```
<?php
$id = $_GET['id'];
$controller = new ElephantController;
$elephants = $controller->read();
```

We will then loop through all of the elephants and check if their ID matches the "id" key which was passed to the Edit Elephant page from the View Elephants page. If so, we will save that elephant object as `$this_elephant`.

```
foreach ($elephants as $elephant_num => $elephant_row):
  if ($elephant_row['id'] == $id):
       $this_elephant = $elephant_row['object'];
  endif;
endforeach;
?>
```

The View file for the Edit Elephant page will have a form on it with an action attribute which points to the address ***"../controllers/routes.php?action=update"*** in order to call the `update()` method in our Routes file.

The name attribute of each input or select field must match with the key we used to retreive the data from the `$_POST` array in our model. The value of each input tag will be filled in with the elephant's currently defined properties by echoing them into the value attribute.

For the select tag we will check each option to see if it matches the value of the elephant's `$image` property using an if statement. If it does, we will echo the `selected` attribute into the tag. We also must have a button tag with a type of "submit".

```
<form action="../controllers/routes.php?action=update" method="post">
  <input type="hidden" name="id" value="<?php echo $id; ?>" id="id">

  <label for="name">Name:</label>
  <input type="text" name="name" value="<?php echo
       $this_elephant->name; ?>" id="name">

  <label for="age">Age:</label>
  <input type="number" name="age" value="<?php echo
       $this_elephant->age; ?>" id="age">

  <select name=”image” id=”image”>
       <option value=”bananas.jpg" <?php if ('bananas.jpg' ==
            $this_elephant->image): echo 'selected'; endif; ?>Bananas
            </option>
       <option value=”beach.jpg"<?php if ('beach.jpg' ==
            $this_elephant->image): echo 'selected'; endif; ?>Beach
            </option>
        <option value="big-ball.jpg"<?php if ('big-ball.jpg' ==
            $this_elephant->image): echo 'selected'; endif; ?>Big
Ball
            </option>
        <option value="birds.jpg"<?php if ('birds.jpg' ==
            $this_elephant->image): echo 'selected'; endif; ?>Birds
            </option>
        <option value="dog.jpg"<?php if ('dog.jpg' ==
            $this_elephant->image): echo 'selected'; endif; ?>Dog
            </option>
    </select>

    <button type="submit"><span class="fa fa-plus"></span>
        Add Elephant</button>
</form>
```