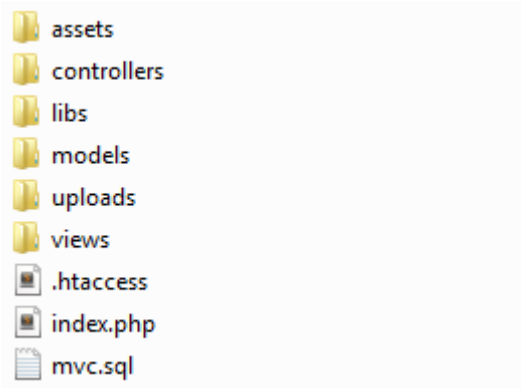
Class 8

MVC

BY:

BRUNO ANGELO MEDEIROS
bruno.medeiros@sc.senai.br

DIRECTORY HIERARCHY



ASSETS DIRECTORY

In here we keep the js, css and media files.

CONTROLLER DIRECTORY

We will use it to store the classes of the system control layer, responsible for intermediating between the data layer and views.

LIBS DIRECTORY

In this directory, we will save classes directly attached to the system, such as data filter classes, generic validations, helpers (if any), interfaces and abstractions not linked to the system's business layer.

MODELS DIRECTORY

In model we will save the data classes directly abstracted and linked to the system's business rules, such as the Contact and Phone classes, among others.

UPLOADS DIRECTORY

The files that were uploaded using the form in the application.

VIEWS DIRECTORY

This is the directory where we will store the system's HTML files, such files represent the view layer.

FILES STRUCTURE

HTACCESS

In this file, we use to modify the default URL that is currently by GET to work like this:

Without htaccess:

```
1.  http://localhost/index.php?q=product/index
```

With htaccess:

```
1.  http://localhost/product/index
```

We call this a friendly URL.

The code for this is:

```
RewriteEngine On
RewriteCond %{REQUEST_FILENAME} !-f
RewriteCond %{REQUEST_FILENAME} !-d
RewriteRule ^(.*)$ index.php?q=$1
```

APP

The APP file is used to start the application, here we treat the URL that will be handled by .htaccess

```
<?php

class App
{
    public function init() {
        $action = 'actionIndex';
        $param = null;

        if (isset($_GET['q'])) {
            $splitted = explode('/', $_GET['q']);
            $controllerName = array_shift($splitted) . 'Controller';
            if (isset($splitted[0])) {
                $action = 'action' . array_shift($splitted);
            }

            if (isset($splitted[0])) {
                $param = array_shift($splitted);
            }

            require_once 'controllers/' . $controllerName . '.php';
            $controller = new $controllerName;
            $controller->{$action}($param);
        }
    }
}

?>
```

MODEL

In the MVC structure for WS we use two different file types for the model, the generic and another specific.

GENERIC

In generic, we use to create the connection structure using PDO. For this we must extend the class PDO that contains the necessary functionalities for the correct operation.

```
<?php

class Model extends PDO
{
    public $attributes = [];

    public function __construct() {
        parent::__construct('mysql:host=localhost;dbname=mvc;charset=utf8', 'root', '', [PDO::ATTR_DEFAULT_FETCH_MODE => PDO::FETCH_OBJ, PDO::ATTR_ERRMODE => PDO::ERRMODE_EXCEPTION]);
    }
}
?>

```

Note:

- **extends** is used to inherit methods from another class, in our case we are inheriting the methods from the PDO to be able to make the connection.
- The **__construct** method of a class serves to perform some behavior (value assignment, method execution, etc.) as soon as an instance of it is created. This brings advantages because it avoids the execution of repetitive and obligatory tasks.

SPECIFIC

And the specific model we use for each entity in our system, for example:

```php
<?php

class Product extends Model
{
    private $id;
    private $name;
    private $description;
    private $table = 'product';

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

    public function setId($id) {
        $this->id = $id;
    }

    public function getName() {
        return $this->name;
    }

    public function setName($name) {
        $this->name = $name;
    }

    public function getDescription() {
        return $this->description;
    }

    public function setDescription($description) {
        $this->description = $description;
    }

    public function insert(){
        $stmt = $this->prepare('INSERT INTO ' . $this->table . '(name,description) VALUES(?,?)');
        return $stmt->execute([$this->name, $this->description]);
    }

    public function update(){
        $stmt = $this->prepare('UPDATE ' . $this->table . ' SET name = ?,description = ? WHERE id = ?');
        return $stmt->execute([$this->name, $this->description, $this->id]);
    }

    public function find($id) {
        $stmt = $this->prepare('SELECT * FROM ' . $this->table . ' WHERE id = ?');
        $stmt->execute([$id]);
        return $stmt->fetchObject(get_called_class());
    }

```

```
50.     public function findAll() {
51.         $stmt = $this->query('SELECT * FROM ' . $this->table);
52.         return $stmt->fetchAll(PDO::FETCH_CLASS, get_called_class());
53.     }
54.
55.     public function delete() {
56.         $stmt = $this->prepare('DELETE FROM ' . $this-
    >table . ' WHERE id = ?');
57.         return $stmt->execute([$this->id]);
58.     }
59. }
60. ?>
```

CONTROLLER

Of the same that there are two types of model, with the controller is the same thing. In the generic we used to create the rendering of the view.

GENERIC

```
1.  <?php
2.
3.  class Controller
4.  {
5.      public function render($view, $data = array()) {
6.          extract($data);
7.          ob_start();
8.          include 'views/' . $view . '.php';
9.          $content = ob_get_contents();
10.         ob_end_clean();
11.         require_once 'views/layouts/main.php';
12.     }
13. }
14. ?>
```

SPECIFIC

In specific, we inherit the methods of the Controller class and we use to create the business rules and send to the view the class instance, in this case we send the product to the view to be manipulated as the user needs.

The example below contains CRUD which consists of listing all products, creating a new product, updating an existing product, and deleting a product.

```
1.  <?php
2.
3.  class ProductController extends Controller
4.  {
5.      public function actionIndex() {
6.          $products = (new Product())->findAll();
7.          return $this->render('product/index', ['products' => $products]);
8.      }
9.
10.     public function actionCreate() {
11.         $product = (new Product());
12.
13.         if(isset($_POST) && !empty($_POST)) {
14.
15.             $name = $_POST['name'];
16.             $description = $_POST['description'];
17.
18.             $product->setName($name);
19.             $product->setDescription($description);
```

```
            $product->insert();

            header("Location: ".URL."product/index");
        }

        return $this->render('product/create', ['product' => $product]);
    }

    public function actionUpdate($id) {
        $product = (new Product())->find($id);

        if(isset($_POST) && !empty($_POST)) {
            $name = $_POST['name'];
            $description = $_POST['description'];

            $product->setName($name);
            $product->setDescription($description);
            $product->update();

            header("Location: ".URL."product/index");
        }

        return $this->render('product/create', ['product' => $product]);
    }

    public function actionDelete($id) {
        $product = (new Product())->find($id);
        $product->delete();

        header("Location: ".URL."product/index");
    }
}
?>
```

AUTOLOAD

This function is used to automatically load classes when called.

```
<?php

function __autoload($className) {
    require_once 'models/' . $className . '.php';
}
?>
```

INDEX.PHP

In index.php we have created a constant variable to capture the URL that will be used later. Then we add autoload.php and initialize it.

```
<?php

define("URL", "http://".$_SERVER['SERVER_NAME'].explode("index.php", $_SERVER[
'SCRIPT_NAME'])[0]);

require_once 'libs/autoload.php';
$app = new App;
$app->init();

?>
```

VIEW

The view is where it contains the largest number of files, because in it we will place each page that the user must access to use the application.

In this MVC pattern created for World Skills, we have the main file, named main.php which is located in the folder layouts within the views folder.

MAIN.PHP

In the main.php contains the layout of the application, we use this way so that we do not have to repeat some elements of the layout.

In our example, we have a simple layout:

```
1.  <!DOCTYPE html>
2.  <html lang="en">
3.  <head>
4.      <meta charset="UTF-8">
5.      <title>MVC Example</title>
6.  </head>
7.  <body>
8.      <?= $content ?>
9.  </body>
10. </html>
```

The **$content** variable is being populated by the controller that was described above.

SPECIFIC VIEWS

The first file, we use to list all of the entity's items.

```
1.  <h1>Products</h1>
2.  <a href="<?= URL; ?>product/create">Create</a>
3.  <table>
4.      <thead>
5.          <tr>
6.              <th>ID</th>
7.              <th>Name</th>
8.              <th>Description</th>
9.              <th>Action</th>
10.         </tr>
11.     </thead>
12.     <tbody>
13.         <?php foreach ($products as $product): ?>
14.             <tr>
15.                 <td><?= $product->getId(); ?></td>
16.                 <td><?= $product->getName(); ?></td>
17.                 <td><?= $product->getDescription(); ?></td>
18.                 <td><a href="<?= URL; ?>product/update/<?= $product-
    >getId(); ?>">Edit</a> | <a href="<?= URL; ?>product/delete/<?= $product-
    >getId(); ?>">Delete</a></td>
19.             </tr>
20.         <?php endforeach; ?>
21.     </tbody>
22.     <tfoot>
23.         <tr>
24.             <th>ID</th>
25.             <th>Name</th>
26.             <th>Description</th>
27.             <th>Action</th>
28.         </tr>
29.     </tfoot>
30. </table>
```

And the other we use to create and edit the items.

```
<h1>Manage Product</h1>

<form method="post" action="#">
    <label for="name">Name</label>
    <input type="text" name="name" id="name" required value="<?= $product->getName() ?>">

    <label for="description">Description</label>
    <textarea name="description" id="description" required><?= $product->getDescription() ?>

<button type="submit">Register</button>
</form>
```