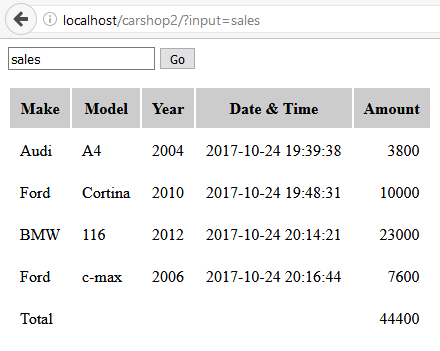
# Homework: HTTP, HTML and PHP

# PHP OOP and MySQL in service of HTTP and HTML

This document defines the homework assignments for ["PHP Web Development Basic" Course @ Software University](https://softuni.bg/trainings/1746/php-web-developmentbasics-september-2017). Please submit ……………………………………………….???

|  |
| --- |
| Important!  **Download** the initial code **12.HTTP-HTML-and-PHP-Initialcode.zip.**  **Download** the initial database SQL dump file **12.HTTP-HTML-and-PHP-Initialsqldump.zip.**  **Import the** SQL into your MySQL server trough a SQL client (like HeidiSQL or a command line tools) |

# Car Sales App for the Web

You are already **familiar** with John Pavlov who has a small business in Eastern Europe for selling old cars. **You have already programmed a CLI app** for him. But he has new staff members who don't want to use CLI but a regular **web browser**. You will use some code and the database from the previous tasks but in different ways.

First you have to **download the code** into a subfolder of your web server (for example in C:\xampp\htdocs\carshop2).

After installation of code you will get something like the **screenshot on the right**.

Then you have to study the code that is already written. You have four subfolders: **controller**, **core**, **model** and **view**. They contain different kind of files. We will start by view.

# The View Folder

Here are all your HTML and CSS files. They may have a .php extension but they have HTML mixed with some PHP. Explore the **header.php**, **footer.php**, **main.php** and **read\_sales.php**

# The Core Folder

This folder contains **two abstract classes** (intended to be extended) which contain very simple functionality.

The abstract **Controller class** takes as argument a DB connection in his constructor and **manages the input from the browser** using a simple **GET method** which **you will study** in the next lecture. He takes the data from the HTML form and inserts it in $this->input.

|  |
| --- |
| **Controller.php**  **<?php abstract class** Controller{    **protected $db** = **null**;  **protected $input** = **null**;    **public function** \_\_construct($db){  $this->**db** = $db;  **if**(**isset**($\_GET[**'input'**])){  $this->**input** = $\_GET[**'input'**];  }  }    **abstract public function** main();  } |

**Controller** contains only one method that should be implemented: main(). So is created a very simple abstraction layer for controlling data input.

The Model class is the other one. It looks like this:

|  |
| --- |
| **<?php abstract class** Model {  **protected $table** = **null**;  **protected $db** = **null**;    **public function** \_\_construct($db){  $this->**db** = $db;  }  } |

Its purpose is to create an **abstraction layer** to work with the **database**. $this->table will hold the **name** of the **DB table** with which a model (a class that works with a particular DB table) will work and also a connection to a new PDO instance ( $this->db ).

# The Model Folder

This folder contains 3 classes. Every class works with a table from the DB. For example:

1. **SalesModel** with **sales** table

2. **CustomersModel** with **customers** table

3. **CarsModel** with **cars** table.

In each of the three you can see that there is a method called create() which inserts a new row into the particular table. Only in SalesModel you have a read() and a readTotal() method which use PDO to read all sales and return the result.

# The Controller Folder

This folder contains the MyController class which does the main job. An instance is created in index.php:

|  |
| --- |
| **<?php class** MyController **extends** Controller {  **public function** main()  {  **include "view/header.php"**;  **include "view/main.php"**;  **switch**($this->**input**){  **case 'sales'**:  $this->viewSales();  **default**:  **break**;  }  **include "view/footer.php"**;  }   **public function** viewSales()  {  $s = **new** SalesModel($this->**db**);  $sales = $s->readAll();  $sales\_total = $s->readTotal();  **include "view/read\_sales.php"**;  }  } |

The main() method loads the **header, main** (input data form) and the **footer** bu using a simple include statements. In case John enters 'sales' it starts the method **getSales()** which uses the **SalesModel** to read all sales from the database.

# The Root Folder

The root folder contains only **db\_config.php** and **index.php**. In db\_config.php a new connection to MySQL is done through PDO and index.php.

|  |
| --- |
| index.php  <?php *// Load DB* include "db\_config.php";  *// Load core classes* include "core/Model.php"; include "core/Controller.php";  *// Load model classes - they extend Model.php* include "model/CarsModel.php"; include "model/SalesModel.php"; include "model/CustomersModel.php";  *// Load Controller class - it extends Controller.php* include "controller/MyController.php"; $main = new MyController($db); $main->main(); |

So index.php loads all classes. It creates an instance of MyController and passes the DB connection. So far this is a rudimentary MVC (Model/View/Controller) application as you can see.

The other simple conclusion is that this is a beginning of a **CRUD** (Create / Read / Update/ Delete) application because these four functions happen in the models.

# Problem 1. Add a Car, Customer and Sale

First thing that John must do is to be able in one line to add both a car and his customer into the database. Because these two go together you need to use a **transaction in your SQL**. The input and output must be like these:

|  |  |
| --- | --- |
| **Input** | **Ouptut** |
| Audi, A4, 2004, Ivan, Ivanov, BGN 7000 | New sale entered 2017-01-24 10:13 |
| BMW, 116, 2010, Ilia, Petrov, BGN 24000 | New sale entered 2017-01-24 10:45 |
| Ford, Focus, 2004, Stoyan, Lazarov, BGN 3800 | New sale entered 2017-01-25 14:34 |

Wrap your code in a AddSale() method in **MyController**. Create a **new html file** for the output with a proper name. The output should appear in a paragraph below the input interface. Use the create() method of **SalesModel**. If needed **add functionality to the model**. If you know CSS put your CSS in view/styles.css and make the output with a blue border.

**Important!** Do not forget that to add a **car** and a **customer** you need to create a new sale and you need **car\_id** and **customer\_id. These are returned by create() methods in their models!**

**Add three new deals as in the example. Take data from sites like mobile.bg. Test your new code by entering 'sales' and see the result.**

# Problem 2. View all Customers

The people in the cars hop need to view their customers. The want by entering 'customers' in the interface to see all their customers in a nice table.

Write the controlling functionality in viewCustomers() in MyController. What new methods do you need in the **CustomersModel?** Create your queries in proper methods in the CustomersModel and use them in your new method viewCustomers(). Put your html in **view/view\_customers.php.**

We have the information in three different tables (cars, customers, sales) but we would like to have it in one abstract table which is a combination of all. Create a new MySQL view called 'deal' which shows: Sale date\_time, sale amount, Car model, car year, customer first\_name, customer family\_name.

# Problem 3. View all Cars

Do the same like in problem 2 but for the cars. Put your control functionality in viewCars(). Choose a proper file name for your HTML file and put it in the proper folder. Create the needed model methods.

# Problem 4. Create an HTML Menu

Instead of using the input create a simple HTML menu on top like this: **Sales | Cars | Customers**

Each of this Is a link to:

<http://localhost/carshop2/?input=sales>

<http://localhost/sql/carshop2/?input=cars>

<http://localhost/sql/carshop2/?input=customers>

To create the link use the HTML tag like:

|  |
| --- |
| <**a href="http://localhost/carshop2/?input=sales"**>Sales</**a**> |

Put your menu in /view/menu.php. **Map all links** to the functionality in problem 1, problem 2 and problem 3! Change the main() method in the appropriate way.

# Problem 5. Create 'Page Not Found' Page

Create a special **/view/page\_not\_found.php** page which should load when the user enters wrong input either in the URL or in the form. Insert some good looking image there. You must also change main() in MyController.

**Return HTTP error code 404** by using header() PHP function. For details read: <http://php.net/manual/en/function.header.php>

# Problem 6. Search a Car & Owner by Make

Create a new method in MyController called searchCarOwner() which on input, searches all cars and returns all car /owner combinations that match the particular make.

|  |  |
| --- | --- |
| **Input** | **Ouptut** |
| Search, Audi | HTML Table of:  Make, Model, Year, Car ID, Owner first name, Owner last name, Customer ID |

Implement in the **CarsModel class** the needed methods. Use an **SQL join** statement. If nothing is found use a file **view/nothing\_found.php** with a comic picture inside. You can use <img src="some\_file.jpg"/> to put a comic image.

# Problem 7. Error page

Create an error page (template, view) which will be displayed if an exception happens

Change the code :

|  |
| --- |
| **print "Error!: "** . $e->getMessage() . **"<br/>"**; |

Display something like the screenshot on the right. The error page should look load a view **error\_page.php**. Change your code to handle all possible exceptions and to load the error page with the error message.

# Problem 8. Create Update Functionality

Many times when information is entered it is mistaken. Create methods in MyController and in the appropriate models to change:

1.Customer first name and family name

2.Car make, model and year

2.Sale amount and time

|  |  |
| --- | --- |
| **Input** | **Output** |
| Update customer, 3, Stoian, Zachariev | Customer with ID 3 updated. |
| Update car, 8, Ford, c-max, 2007 | Car with ID 8 updated |
| Update sale, 3, 2017-10-24 20:14:21, 23100 | Sale with ID 3 updated |

# Problem 9. Develop the Interface\*

Until now the app uses a very simple interface. Come up with a better interface in which the 'Go' button will become ' Search ' and will be able to search in cars, customers and sales. Use MySQL comparison functions to do this:

<https://dev.mysql.com/doc/refman/5.7/en/string-comparison-functions.html>

Create your own templates with HTML forms to:

1.Update customer information

2.Update car information

3.Enter a new sale (problem 1)

# Problem 10. Develop the Framework\*\*

How can you develop further the framework of **controller** and **model** to:

1. Use a loadView() method instead of include for the HTML templates. Distribute the different template files in appropriate sub-folders.

2. Be able to work with **more than one controller**. How would it look with SalesController, CarsController and CustomersController? Code some **routing scheme** like:

http://localhost/sql/carshop2/?controller=SalesController&action=viewSales&input=....

3.**Distribute your code** in different methods to fit the routing scheme in which every $\_GET['controller'] chooses a different class (SalesController, CarsController and CustomersController) and each $\_GET['action'] chooses a different method /function in the class (like viewSales() )

Code the routing scheme in the **core functionality**.