L-Università ta' Malta
Faculty of Information & Communication Technology
Department of Computer Information Systems

Principles of Web Application Architecture Assignment

Matthias Bartolo* (0436103L), Jerome Agius* (0353803L), Isaac Muscat* (0265203L)
*Bachelor of Science in Information Tech (Hons) Artificial Intelligence

---

Study-unit: **Principles of Web Application Architecture**
Code: **CIS1054**
Lecturer: **Dr Chris Porter**

Plagiarism Form

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

Declaration

Plagiarism is defined as "the unacknowledged use, as one's own work, of work of another person, whether or not such work has been published" (Regulations Governing Conduct at Examinations, 1997, Regulation 1 (viii), University of Malta).

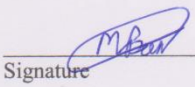
I/ We*, the undersigned, declare that the [assignment / Assigned Practical Task report / Final Year Project report] submitted is my / our* work, except where acknowledged and referenced.

I/ We* understand that the penalties for making a false declaration may include, but are not limited to, loss of marks; cancellation of examination results; enforced suspension of studies; or expulsion from the degree programme.

Work submitted without this signed declaration will not be corrected, and will be given zero marks.

* Delete as appropriate.

(N.B. If the assignment is meant to be submitted anonymously, please sign this form and submit it to the Departmental Officer separately from the assignment).

Matthias Bartolo Student Name	 Signature
Isaac Muscat Student Name	IMuscat Signature
Jorge Agius Student Name	J. Agius Signature
Student Name	Signature

CIT 1054 Course Code	Principles of web application architecture Assignment Title of work submitted

26/05/2022
Date

Contents

1. Part 1 – Setup and Preliminary Tasks

Link to GitHub repository: link

Within the README file, please find a step-by-step guide to set-up the website.

Note that within the files, comments are added to further explain the code.

1.1 Setting up HTTP server with PHP support

When it comes to setting up an HTTP server with PHP support one can make use of XAMPP which is simple to install and make use of. To install XAMPP one can go to this link and select the appropriate download option. Upon concluding the download one can simply follow the on-screen wizard instructions to complete the XAMPP installation.

XAMPP provides a multitude of facilities which one may opt to make use of such as an Apache server to be able to host your web pages and access them through a browser. PHP support can be installed separately although it comes packaged with XAMPP.

Although XAMPP comes packaged with a working Apache server that supports PHP, a freshly downloaded Apache server would require setting up to facilitate the use of PHP. In this case the user would first need to have access to an Apache server and then install PHP functionality on it. This can simply be done by going to the terminal and typing "sudo apt install php". Then, the required modules would also need to be installed. This can also be done through a simple command, this being "sudo apt install libapache2-mod-php". Finally, the user would have to enable the module using the command "sudo a2enmod php" followed by a server restart.

After the installation and setup has been completed, the user would now be able to pass .php documents to the server which would be identified by it as PHP code and thus, the server would pass it to the PHP interpreter which would execute all the relative PHP code and then the page would be loaded and shown to the user. The user would need to place the PHP files in the XAMPP/htdocs directory.

XAMPP also comes equipped with MySQL, which is a relational database system. This feature in conjunction with the Apache web server and PHP scripting language, offers data storage for web services.

XAMPP also come packaged with Mercury, which is a local mail server, which allows sending of emails from the server. However, we opted not to use Mercury and instead we utilised PHPMailer.

These components can be seen in Figure 1.1

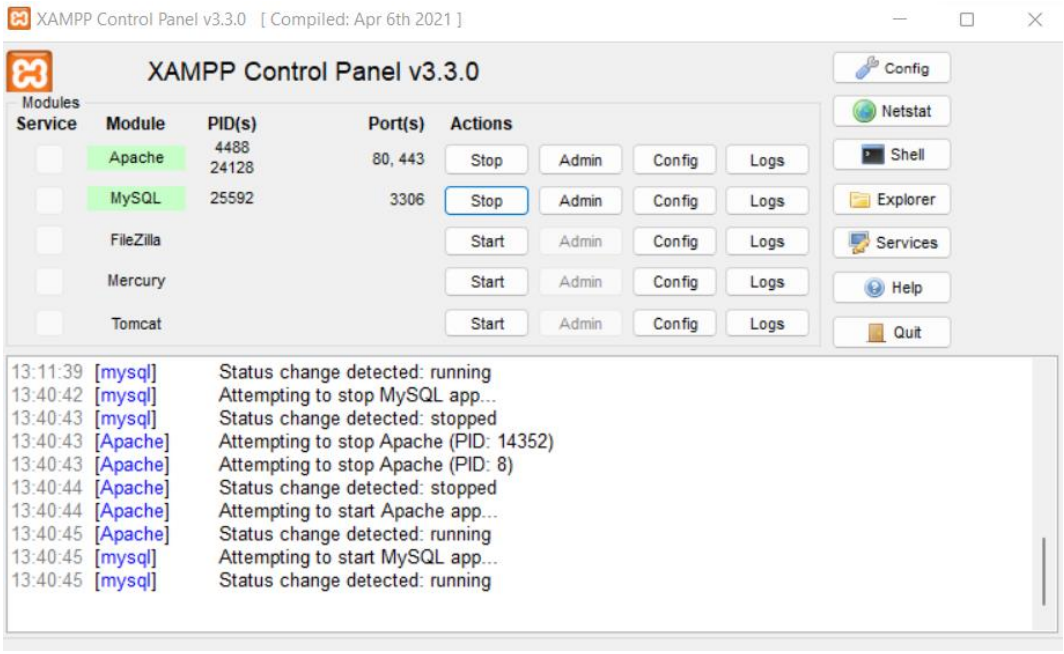


Figure 1.1 – XAMPP terminal

1.2 serverdt.php

The approach taken in solving this problem is as follows:
Firstly, a session is started, and the current date and time are obtained in the format 'd/m/Y h:i:s a'. This was facilitated through the date and time functions. The date was then stored in the session variable time.

Finally, the session variable's contents were displayed by using the echo command. The session is terminated if all instances of the website are closed. This can be seen in Figure 1.2.

```php
<?php
// Starting a session
session_start();
// Getting and storing current date and time
$_SESSION['time'] = date('d/m/Y h:i:s a', time());
// printing the contents of the data variable on screen
echo $_SESSION['time'];
?>
```

Figure 1.2 – PHP used in serverdt.php

1.3 lastvisit.php

The approach taken in solving this problem is as follows:
Firstly, a session was started, which was then followed by a conditional statement to check if a session variable called Date was set. In the case that it was not set, hence the user visited the website for the first time, it would retrieve the current date and time and store it in session variable Date.

Following this, the message “This is your first time visiting the page” is displayed using the echo command. In the case that the session variable Date was already set, it would convert the session variable Date’s contents to time and then store the contents in the variable timeBefore.

Similarly, the current data and time was retrieved, converted to time by using the strtotime function and then stored in the variable timeNow. The difference between timeNow and timeBefore was calculated and then stored in the variable duration.

Finally, a message is displayed which shows the contents of the variable duration. The session is terminated if all instances of the website are closed. This can be seen in Figure 1.3 and Figure 1.4.

```php
<?php
// Starting a session
session_start();
?>
```

Figure 1.3 – PHP used in lastvisit.php

```php
<?php
if(isset($_SESSION['Date']))
{
    // Retrieves the time stored in the session variables from the previous page instance and converts it to seconds
    $timeBefore = strtotime($_SESSION['Date']);
    // Retrieves the current time and converts it to seconds
    $timeNow = strtotime(date('Y/m/d H:i:s'));
    // Works out the difference in seconds between the two times
    $duration = $timeNow - $timeBefore;
    // Displays a message
    echo "You first used this page ". $duration . " seconds ago";
}
else
{
  // Sets the time session variable the first time the page is visited
  $_SESSION['Date'] = date('Y/m/d H:i:s');
  echo "This is your first time visiting the page";
}
?>
```

Figure 1.4 – PHP used in lastvisit.php

1.4 processrequest.php

The approach taken in solving this problem is as follows:
Firstly, a session was started. Depending on the form method used (POST / GET), the variables fnameP / fnameG and fageP / fageG would be checked to make sure they were set. If fnameP and fageP were set, and the request method is POST, the variable fnameP was stored in fname whilst fageP was stored in fage. Similarly, if the variables fnameG and fageG were set, and the request method is GET, the variable fnameG was stored in fname whilst fageG was stored in fage.

When the form was submitted, the website was reloaded such that the data that was in the form was placed in their respective variables, hence that is why PHP_SELF was used. This can all be seen in Figure 1.5 and Figure 1.6 and Figure 1.7.

Figure 1.5 – PHP used in processrequest.php

```
<form method="post" action="<?php echo $_SERVER['PHP_SELF'];?>">
  Name: <input type="text" name="fnameP"><br>
  Age: <input type="number" name="fageP"><br>
  <input type="submit">
</form>

<!--Form dealing with get requests-->
<h2>Get Method</h2>

<form method="get" action="<?php echo $_SERVER['PHP_SELF'];?>">
  Name: <input type="text" name="fnameG"><br>
  Age: <input type="number" name="fageG"><br>
  <input name = "submit" type="submit">
</form>
```

Figure 1.6 – Forms used in processrequest.php

```
<?php
if (isset($_POST['fnameP']) and isset($_POST['fageP']) and $_SERVER["REQUEST_METHOD"] == "POST") {
  // collect value of input field
  $_SESSION['fname'] =  $_POST['fnameP'];
  $_SESSION['fage'] = $_POST['fageP'];
}
elseif (isset($_GET['fnameG']) and isset($_GET['fageG']) and $_SERVER["REQUEST_METHOD"] == "GET") {
  // collect value of input field
  $_SESSION['fname'] = $_GET['fnameG'];
  $_SESSION['fage'] = $_GET['fageG'];
}
?>
```

Figure 1.7 – PHP used in processrequest.php

1.5 readsession.php

The approach taken in solving this problem is as follows:
Firstly, a session was started. If the session variables fname and fage were empty, a message was displayed stating that “The Name or Age session parameters are not set”. Otherwise, a message was displayed by retrieving the session variables fname and fage. This can be seen in Figure 1.8.

```
if (empty($_SESSION['fname']) or empty($_SESSION['fage'])) {
  echo "<u>(Session Variables)</u><br> The U_Name or U_Age session parameters are not set";
}
else{
  echo "<u>(Session Variables)</u><br> Username: " . $_SESSION['fname'] . ", " . "Age: " . $_SESSION['fage'];
}
```

Figure 1.8 – PHP used in readSession.php

A session is created every time the user visits the website and there is no session currently active. Each user is given an id which is stored in a cookie. This is used to identify the respective session for each user.

Whenever a request is issued for a session variable the user id which is stored as a cookie is used to check which variables the user has access to. Then the server returns the respective session variables based on the user id. By default, a session lasts until the user closes their browser, but this can be changed from within the php.ini file on the web server.

2. Part 2 – Dynamic Web Application

Link to GitHub repository: link

Since it is considered a best practice to separate PHP code from HTML, our project opts to make use of TWIG, which is a templating engine used to do such a task. Thus, each page section will be separated into two parts, the PHP and HTML sections.

2.1 Header Page

The header.html page consists of two navigation bars, one for desktop screens and the other for mobile devices. Both navigation bars utilise a similar code structure, as seen in Figure 2.1.

```
<nav class = "Desktop-Nav sticky">
  <ul class = "nav-list">
    <li class="nav-elem"> <a class="nav-link" href="main.php">Home</a></li>
    <li class="nav-elem"> <a class="nav-link" href="contact.php">Contact Us</a></li>
    <li class="nav-elem"> <a class="nav-link" href="menu.php">Menu</a></li>
    <li class="nav-elem"> <a class="nav-link" href="favourites.php">Favourites</a></li>
    <li class="nav-elem"> <a class="nav-link" href="workers.php">Workers</a></li>
  </ul>
</nav>
```

Figure 2.1 – Desktop Navigation Bar

Through CSS rules, the website will switch between the two navigation bars according to the screen's size as seen in Figure 2.2. When the screen size is greater than 426 pixels, the desktop navigation bar is visible, whilst the display property of the mobile navigation bar is set to none. On the other hand, if the screen size is less than 426 pixels, the opposite applies.

```
/*Hides Mobile Nav Bar*/
@media (min-width: 426px) {
  .Mobile-Nav{
    display: none;
    position: absolute;
  }
}

/*Hides Desktop Nav Bar*/
@media (max-width: 426px) {
  .Desktop-Nav{
    display: none;
  }
```

Figure 2.2 – Navigation Bar CSS

The following JavaScript was used to implement the burger menu dropdown functionality in the mobile nav bar.

```
function navFunction() {
  var x = document.getElementById("myLinks");
  var y = document.getElementById("link-container");
  var light_blue = "rgb(92, 154, 209)";
  if (x.style.display === "block") {
    y.style.border = "black 0px solid";
    x.style.display = "none";
  } else {
    x.style.display = "block";
    y.style.background = light_blue;
    y.style.width = "50vw";
  }
};
```

Figure 2.3 – Burger Menu Dropdown Functionality

Figure 2.4 and Figure 2.5 depict the navigation bars implemented.

Figure 2.4 – Desktop Navigation Bar

Figure 2.5 – Mobile Navigation Bar

2.2 Footer Page

The footer.html page consists of three links to three respective social media accounts. Since our website was inspired by Los Pollos Hermanos, we opted to provide links to their social media accounts, as seen in Figure 2.6 and Figure 2.7.

```
<!-- Displaying Social Media Links and Authors-->
<footer>
    <ul class = "footer-list">
      <li class = "footer-elem"><a class = "footer-link" tabindex="0" href="https://www.facebook.com/Los.Pollos.Hermanos.Inc" title="facebook"><i class="fa fa-brands fa-facebook fa-4x"></i></a></li>
      <li class = "footer-elem"><a class = "footer-link" tabindex="0" href="https://twitter.com/eatlospollos" title="twitter"><i class="fa fa-twitter fa-4x"></i></a></li>
      <li class = "footer-elem"><a class = "footer-link" tabindex="0" href="https://www.instagram.com/eatlospolloshermanos/" title="instagram"><i class="fa fa-brands fa-instagram fa-4x"></i></a></li>
    </ul>
    <div class="footer flexCenter">
      <p tabindex="0" class="footer-text">
        CIS1054 - Principles of Web Application Architecture</p>
      <p tabindex="0" class="footer-text">
        Jerome Agius, Matthias Bartolo and Isaac Muscat</p>
    </div>
</footer>
```

Figure 2.6 – Footer Links Code

Figure 2.7 – Footer Links in Web Page

2.3 Layout Page

The layout.html page is used as a template for each web site so as to avoid repeated code. It consists of the code within the head tag which will be used for each individual page. Note that the title has to be passed for each page so that it is unique. The code depicted above in the header.html and footer.html files respectively is included in its appropriate location in the template. It also includes the masthead section, which is universal for each page and based on the mastheadImg parameter passed via TWIG, which will determine the masthead image to be used. The MTitle and MSTitle hold the main title and any subtext which should be displayed on the masthead. Finally, there is also the content block which is used to denote where any page specific HTML will be placed in the template, as seen in Figure 2.8. Examples of how the masthead changes depending on the web page can be seen in Figure 2.9 and Figure 2.10.

```
<!DOCTYPE html>
<html>
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
  <meta name="description" content="CIS1054-DYNAMIC_WEB_APPLICATION" />
  <meta name="author" content="Jerome Agius/Matthias Bartolo/Isaac Muscat" />
  <title>{{ title }}</title>
  <link rel="icon" type="image/x-icon" href="../images/logo.ico" />
  <script src="https://kit.fontawesome.com/20e8212d54.js" crossorigin="anonymous"></script>
  <!-- Core theme CSS -->
  <link href="../css/main.css" rel="stylesheet" />
  <script src="../js/main.js"></script>
</head>

<body>
  {% include 'header.html' %}
  <!-- Displaying the page Title, Sub title and respective image-->
  <section class = "sec-masthead">
    <header class ="masthead {{mastheadImg}}">
      <h1 tabindex="0" class="headertext">{{MTitle}}</h1>
      <h2 tabindex="0" class="subheadertext">{{MSTitle}}</h2>
    </header>
  </section>

  {% block content %}
  {% endblock %}

  {% include 'footer.html' %}
</body>
</html>
```

Figure 2.8 – Layout Page Code

Figure 2.9 – Masthead Example 1

Figure 2.10 – Masthead Example 2

2.4 Home Page

The main.html page consists of the header and footer which were implemented in the layout.html template (described above), as well as three pairs of images accompanied by paragraphs which display basic information about the restaurant, as can be seen in Figure 2.11 and Figure 2.12.

```
<li class = "info-elem">
    <img tabindex="0" class = "info-img" src = "../images/generic-img2.jpg" alt = "Los Pollos Hermanos restaurant" title = "Los Pollos Hermanos restaurant">
    <p tabindex="0" class = "info-par">Back in 2011, Gus and Saul worked in a chicken-themed restaurant.
      However, things didn't quite work out for them as the restaurant closed down at the end of the year.
      This did not stop the brothers from doing what they love though.
      After months of planning, they opened our first restaurant in San Giljan.
      After the immediate success of the first restaurant, Los Pollos Hermanos was no longer a restaurant, but rather a chain.
      Since then we have hired class chefs which accompanied by Gus and Saul, provide flavour-rich food.</p>
 </li>
```

Figure 2.11 – Code used to display Image accompanied by Paragraph

Figure 2.12 – Image and Paragraph as seen in the Home Page

The main page also includes two information sections which depict the opening hours as well as the addresses of the restaurants. To retrieve said information, a TWIG for loop was used to loop through the contents of a database table and display the information as required. Access to the database was facilitated through the code present in the main.php file, which is discussed in the following section. The above can be seen in Figure 2.13 and Figure 2.14.

```
<!--Showing working hours-->
<div class = "open-contact">
  <div class = "info-open-container">
    <ul class = "info-open-list main-list-style">
        <li tabindex="0" class = "list-elem underline remove_p"><h3>Opening hours</h3></li>
      {% for tempOpHours in openingHours %}
        {% if (tempOpHours.id % 2) == 0 %}
          <li tabindex="0" class = "list-elem chicken_p">{{ tempOpHours.day }}: {{ tempOpHours.time }}</li>
        {% else %}
          <li tabindex="0" class = "list-elem fries_p">{{ tempOpHours.day }}: {{ tempOpHours.time }}</li>
        {% endif %}
      {% endfor %}
    </ul>
  </div>
</div>

<!--Showing Restaurant addresses-->
<div class = "address">
  <div class = "address-container">
    <ul class = "address-list main-list-style">
      <li tabindex="0" class = "list-elem underline remove_p"><h3>Malta Location Addresses</h3></li>
      {% for tempLocation in locations %}
        <li tabindex="0" class = "list-elem beer_p">{{ tempLocation.address }}</li>
      {% endfor %}
    </ul>
  </div>
</div>
```

Figure 2.13 – Working Hours and Addresses Code

Figure 2.14 – Working Hours and Addresses as seen in the Home Page

Finally, the following PHP code was used in conjunction with the main.html file to construct the home page. The main.php file includes a header which is used to denote the cache functionality which is set to public as well as having a time limit of one day. Furthermore, it is accessing the tables locations and opening_hours from the database 'restaurantinfo' and retrieving all of their records. Finally, TWIG is being used to render the PHP page whilst using the HTML file as a template. The corresponding parameters are passed as outlined in the layout section alongside the retrieved records as seen in Figure 2.15.

```php
<?php
  //Acquiring bootstrap.php and database.php
  require_once __DIR__.'/bootstrap.php';
  require_once __DIR__.'/database.php';

  //Setting Cache Control
  header("cache-control: public, max-age=86400");

  //Get the db object
  $db = new Db();

  //Performing a query to retrieve all the records from the locations table
  $locations = $db -> select("SELECT * FROM locations");
  //Performing a query to retrieve all the categories from the opening_hours table
  $openingHours = $db -> select("SELECT * FROM opening_hours");

  //Passing necessary parameters to html template to render the page
  echo $twig->render('main.html', array("title" => "Main Page",
  "mastheadImg" => "masthead-main",
  "MTitle" => "Welcome to Los Pollos Hermanos",
  "MSTitle" => "It's finger lickin good",
  "locations" => $locations,
  "openingHours" => $openingHours));
?>
```

Figure 2.15 – PHP used in the Home Page

2.5 Workers Page

The workers.html page consists of the header and footer which were implemented in the layout.html template (described above), as well as three pairs of images accompanied by paragraphs which display information about each worker. To retrieve said information, a TWIG for loop was used to loop through the contents of a database table and display the information as required. Access to the database was facilitated through the code present in the workers.php file, which is discussed in the following section. The above can be seen in Figure 2.16 and Figure 2.17.

```
<!--Showing basic information of the people running the restaurant accompanied by the employee picture-->
<section class = "sec-content">
  <div class = "worker-info-container">
    {% for tempWorker in workers %}
    <div class = "worker-profile-container flexCenter">
      <img tabindex="0" class = "workerPFP" src = "{{ tempWorker.pfp }}" alt = "{{ tempWorker.alt_text }}" title = "{{ tempWorker.name }}">
      <p tabindex="0" class = "info-style">
        Name: {{ tempWorker.name }}<br>
        Position: {{ tempWorker.position }}<br>
        Period Working: {{ tempWorker.period }} years<br>
        Quote: {{ tempWorker.quote }}
      </p>
    </div>
    {% endfor %}
</section>
```

Figure 2.16 – Code used to implement Image accompanied by paragraph of Worker

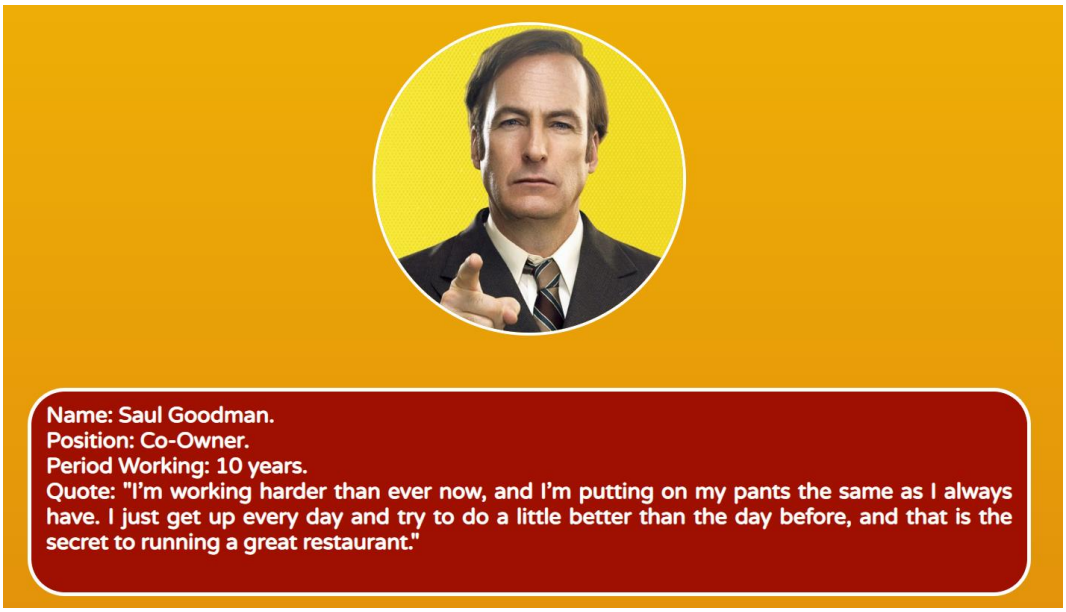


Figure 2.17 – Image and Worker paragraph as seen in the Workers Page

Finally, the following PHP code was used in conjunction with the workers.html file to construct the workers page. The workers.php file includes a header which is used to denote the cache functionality which is set to public as well as having a time limit of one day. Furthermore, it is accessing the table worker_info from the database 'restaurantinfo' and retrieving all of its records. Finally, TWIG is being used to render the PHP page whilst using the HTML file as a template. The corresponding parameters are passed as outlined in the layout section alongside the retrieved records as seen in Figure 2.18.

```php
<?php
  //Acquiring bootstrap.php
  require_once __DIR__.'/bootstrap.php';
  require_once __DIR__.'/database.php';

  //Setting Cache Control
  header("cache-control: private, max-age=86400");

  //Get the db object
  $db = new Db();

  //Performing a query to retrieve all the records from the locations table
  $workers = $db -> select("SELECT * FROM worker_info");

  //Passing necessary parameters to html template to render the page
  echo $twig->render('workers.html', array("title" => "Workers",
  "mastheadImg" => "masthead-workers",
  "MTitle" => "Workers",
  "MSTitle" => "Meet the crew",
  "workers" => $workers));

?>
```

Figure 2.18 – PHP used in the Workers Page

2.6 Menu Page

The menu.html page consists of the header and footer which were implemented in the layout.html template (described above), as well as category names accompanied by their respective food items. The code loops through each food category and displays its name along with its respective food items. To do this, for each category it loops through every food item and checks if its category id matches that of the current category iteration. If this is the case, it will display the food item's name as well as its respective price. The item's name also links to the food item's respective dish details page, explained further below. The above can be seen in Figure 2.19 and Figure 2.20. Whenever a cursor is placed over a food item name, the colour of the text will change appropriately to show that it is clickable.

```
<!--Looping through every Category Row-->
{% for Categories in foodCategories %}
<div class = "flexCenter">
<table class = "menu-info">
  <tbody>
    <tr>
      <!--Displaying Category name-->
      <th class = "menu-header"><a tabindex="0">{{ Categories.categoryName }}</a></th>
    <tr>
  <!--Looping through every food item Row -->
  {% for food in foodItems %}
    <!--Checking if the food item categoryId corresponds to the Categories' categoryID-->
      {% if food.categoryId == Categories.categoryID %}
        <!--Displaying relevant attributes from food item Row-->
        <tr class = "table-data italics">
          <!--The food item name, will serve as a link to the dishDetails.php page, with the item's Id as a parameter, when clicked-->
          <td><a class="menu-link" tabindex="0" href="dishDetails.php?requiredItemId={{ food.itemId }}">{{ food.itemName }}</a></td>
          <td class = "align-right"><a tabindex="0">€ {{ food.itemPrice }}</a></td>
        </tr>
      {% endif %}
  {% endfor %}
  </tbody>
</table>
{% endfor %}
```

Figure 2.19 – Code used to display the Menu Page

Figure 2.20 – One category and its respective Items displayed in the Menu Page

Finally, the following PHP code was used in conjunction with the menu.html file to construct the menu page. Furthermore, TWIG is being used to render the PHP page whilst using the HTML file as a template. The corresponding parameters are passed as outlined in the layout section. This code retrieves all the records in the database's tables, these being the category information as well as the food items, and passes them to the menu.html file, as seen in Figure 2.21.

```php
<?php
  //Acquiring bootstrap.php and database.php
  require_once __DIR__.'/bootstrap.php';
  require_once __DIR__.'/database.php';

  //Get the db object
  $db = new Db();

  //Performing a query to retrieve all the records from the items table
  $items = $db -> select("SELECT * FROM items");
  //Performing a query to retrieve all the categories from the menu table
  $categories = $db -> select("SELECT * FROM menu");
  //Passing necessary parameters to html template to render the page
  echo $twig->render('menu.html', array("title" => "Menu",
  "mastheadImg" => "masthead-menu",
  "MTitle" => "LOS POLLOS HERMANOS MENU",
  "MSTitle" => "Choose your Meal!",
  'foodItems' => $items,
  'foodCategories' => $categories));
?>
```

Figure 2.21 – PHP used in the Menu Page

2.7 Dish Details Page

The dishDetails.html page consists of the header and footer which were implemented in the layout.html template (described above), as well as an interactive image with text displaying information about the selected food item. The code loops through each food category and checks if the selected item's category id matches that of the current iteration. If they match, the respective food item's information is displayed. This information consists of the item's name, description and price as well as a grayscale image of it in the background which when hovered upon, is displayed in colour. The item also contains a heart button which if clicked, either adds or removes the food item from the favourites list. When clicked on, the addFavourites.php function is executed, and hence it will remove or add the item to the favourites session array as necessary. The state of the heart is altered using a JavaScript function. The above can be seen in Figure 2.22 and Figure 2.23.

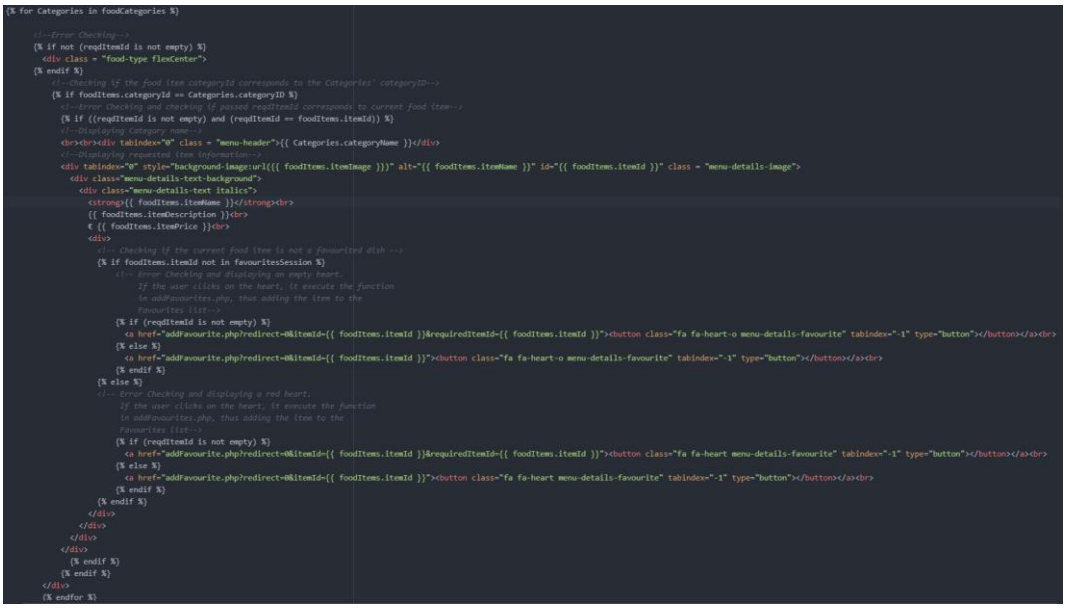


Figure 2.22 – Code used to display Dish Details Page

```
function heartToggle() {
  var x = document.getElementById(arguments[0]);
  if (x.classList.contains("fa-heart-o")) {
    x.classList.remove("fa-heart-o");
    x.classList.add("fa-heart");
  } else if (x.classList.contains("fa-heart")) {
    x.classList.remove("fa-heart");
    x.classList.add("fa-heart-o");
  }
};
```

Figure 2.23 – JavaScript function used to toggle the Heart Symbol

The function above heartToggle() will fill in the heart if it was previously empty and empty the heart if it was previously filled in.

Finally, the following PHP code was used in conjunction with the dishDetails.html file to construct the dish details page. Furthermore, TWIG is being used to render the PHP page whilst using the HTML file as a template. The corresponding parameters are passed as outlined in the layout section. Initially, a query is being issued to the database to retrieve the record's information whose item id matches the GET variable requiredItemId. It also retrieves all category information and passes it to the dishDetails.html. In the case that an error occurs, the user is redirected to the error.html page. The above can be seen in Figure 2.24 and Figure 2.25.

```php
<?php
  session_start();
  //Acquiring bootstrap.php and database.php
  require_once __DIR__.'/bootstrap.php';
  require_once __DIR__.'/database.php';

  //If the session array favourites is not set, it will set declare it as an array
  if (!isset($_SESSION['favourites'])) {
    $_SESSION['favourites'] = array();
  }

  //If statement will only execute if requiredItemId was set
  if(isset($_GET['requiredItemId'])) {

      //Get db object
      $db = new Db();
      //quote used to make the requiredItemId safe for querying
      $menuId = $db -> quote($_GET['requiredItemId']);
      //Performing a query to retrieve all the categories from the menu table
      $categories = $db -> select("SELECT * FROM menu");
      //Performing a query to retrieve the required record which's menuId corresponds to the record's itemId
      $result = $db -> select("SELECT a.*, t.categoryName as type FROM items a inner join menu t on a.categoryId = t.categoryID  WHERE a.itemId=". $menuId);

      //If statement will execute if at least 1 record was retrieved
      if (count($result) > 0){
          //Loading data from result and storing into items
          $items = [
                  'itemId'              => $result[0]['itemId'],
                  'itemName'            => $result[0]['itemName'],
                  'itemPrice'           => $result[0]['itemPrice'],
                  'itemDescription'     => $result[0]['itemDescription'],
                  'itemImage'           => $result[0]['itemImage'],
                  'categoryId'          => $result[0]['categoryId'],
          ];
          //Passing necessary parameters to html template to render the page
          echo $twig->render('dishDetails.html', array("title" => "Dish Details",
            "mastheadImg" => "masthead-dish-details",
            "MTitle" => "Los Pollos Dish Details",
            "MSTitle" => "More Information",
            'foodItems' => $items,
            'foodCategories' => $categories,
            'reqdItemId'=>$_GET['requiredItemId'],
            'favouritesSession' => $_SESSION['favourites']));
       }
      //An error occured, and will redirect the user to the error page
      else
          echo $twig->render('error.html',array("title" => "Error 404","PageImgClass" => "errorPageImage","message" => "The page you requested could not be found."));
    //An error occured, and will redirect the user to the error page
  } else
      echo $twig->render('error.html',array("title" => "Error 404","PageImgClass" => "errorPageImage","message" => "The page you requested could not be found."));
?>
```

Figure 2.24 – PHP used in the Dish Details Page

Figure 2.25 – Food Item displayed in the Dish Details Page (Empty Heart + Filled In Heart)

2.8 Error Page

The error.html page consists of an image as well as an error message. There are two possible messages, "The page you requested could not be found." in the case that a 404 error occurs and "Forbidden Access." In the case that a 403 error occurs. Following the message is a link which leads back to the home page. The background image also changes depending on the screen's size. The above is displayed in Figure 2.26.

```
    <!--Displaying respective error message and image, and redirection link to the main page-->
    <div class="headingError">
      <h1 tabindex="0"><u>{{title}}</u></h1>
      <h2 tabindex="0" class = "EText">{{message}}</h2>
      <a href="http://{{ip}}/ProjectPart2/web/main.php"><button id="HomeButton" type="button">Go Back To Home Page!</button></a>
    </div>
  </body>
  {% endblock %}
```

Figure 2.26 – Code used to display Error Page.

Furthermore, the following PHP code was used in conjunction with the error.html file to construct the error page. Furthermore, TWIG is being used to render the PHP page whilst using the HTML file as a template. The corresponding parameters are passed. The error.php file includes a header which is used to denote the cache functionality which is set to public as well as having a time limit of one day. Depending on the error type, it passes the corresponding error message. The above can be seen in Figure 2.27.

```
<?php
  //Acquiring bootstrap.php
  require_once __DIR__.'/bootstrap.php';
  //Setting Cache Control
  header("cache-control: public, max-age=86400");

  //Retrieving server ip/name
  $ip = $_SERVER['SERVER_NAME'];

  /*Depending on the type of error, the necessary parameters are passed to html template to render the page*/
  if(http_response_code() == 404){
    echo $twig->render('error.html',array("title" => "Error 404","PageImgClass" => "errorPageImage","message" => "The page you requested could not be found.", "ip" => $ip));
  }
  elseif(http_response_code() == 403){
    echo $twig->render('error.html',array("title" => "Error 403","PageImgClass" => "errorPageImage","message" => "Forbidden Access.", "ip" => $ip));
  }

?>
```

Figure 2.27 – PHP used in the Error Page

Also, CSS is used to change the image displayed depending on the screen's size. This can be seen in Figure 2.28 and Figure 2.29.

```
@media screen and (max-width: 1023px) and (min-width: 426px) {
  .errorPageImage {
    background-image: url("../images/error404Medium.png");
  }
```

Figure 2.28 – Error Page Image CSS

Figure 2.29 – Error Page using Medium Screen Size

The errorLayout.html page is used as a template for the error page. It consists of the code within the head tag which will be used for the page. Note that the title is being passed. The content block is used to denote where the extra code will be placed in the template, as seen in Figure 2.30.

```
<!--This is the template layout for error pages that will be used by twig-->
<!DOCTYPE html>
<html class="{{PageImgClass}}">
<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
  <meta name="description" content="CIS1054-DYNAMIC_WEB_APPLICATION" />
  <meta name="author" content="Jerome Agius/Matthias Bartolo/Isaac Muscat" />
  <title>{{ title }}</title>
  <link rel="icon" type="image/x-icon" href="http://{{ip}}/ProjectPart2/images/logo.ico" />
  <script src="https://kit.fontawesome.com/20e8212d54.js" crossorigin="anonymous"></script>
  <!-- Core theme CSS -->
  <link href="http://{{ip}}/ProjectPart2/css/main.css" rel="stylesheet"/>
  <script src="http://{{ip}}/ProjectPart2/js/main.js"></script>
</head>

<body>
  {% block content %}
  {% endblock %}
</body>
</html>
```

Figure 2.30 – Code used to create a layout for the error page

2.9 Favourites Page

The favourites.html page consists of the header and footer which were implemented in the layout.html template (described above), as well as interactive images with text displaying information about the favourited food items. The code loops through each food category and displays its name. For each iteration, a check variable is set to 0. Within each iteration of the loop, the code loops through each food item and checks if the current food item's id matches any food item's id present in the favourites session array. If they match, the respective food item's information is displayed, and the check variable is set to 1. This information consists of the item's name, description and price as well as a grayscale image of it in the background which when hovered upon, is displayed in colour. The item also contains a filled in heart button which if clicked upon, calls addFavourites.php, and hence removes the food item from the favourites list. In the case that the check variable is still 0 after iterating through each food item, a message stating "No items found in this category." is displayed. The above can be seen in Figure 2.31 and Figure 2.32.

```
<!--Looping through every Category Row-->
{% for Categories in foodCategories %}
    <!--Displaying Category name-->
    <div tabindex="0" class = "menu-header">{{ Categories.categoryName }}</div>
    <div tabindex="0" class = "food-type flexCenter">

      <!-- For each iteration of the loop setting the variable check to 0 -->
      {% set check = 0 %}
      <!--Error checking to check whether foodItems exist -->
      {% if foodItems is not empty %}
        <!--Looping through every food item Row -->
        {% for food in foodItems %}
            <!-- Checking if the current food item is a favourited dish and also checking if
            the current food item id corresponds to the current categoryId-->
            {% if food.itemId in favouritesSession and food.categoryId == Categories.categoryID %}
            <!--Setting check to 1, since a favourited item was found-->
              {% set check = 1 %}
              <!--Displaying current item information-->
                <div style="background-image:url({{ food.itemImage }})" alt="{{ food.itemName }}" id="{{ food.itemId }}" class = "menu-details-image">
                  <div class="menu-details-text-background">
                    <div class="menu-details-text italics">
                      <strong>{{ food.itemName }}</strong><br>
                    {{ food.itemDescription }}<br>
                    € {{ food.itemPrice }}<br>
                    <div>
                      <!--Button to remove from favourites list -->
                      <a href="addFavourite.php?redirect=1&itemId={{ food.itemId }}"><button class="fa fa-heart menu-details-favourite" tabindex="-1" type="button"></button></a><br>
                    </div>
                  </div>
                </div>
            </div>
            {% endif %}
        {% endfor %}
        <!--Displaying relevant message if no favourited items were found in the Category-->
        {% if check == 0 %}
            <p class="menu-details-text italics">No items found in this category.</p>
        {% endif %}
          <!--Resetting check to 0-->
          {% set check = 0 %}
        </div>
        <br><br>
        {% endif %}
        {% endfor %}
```

Figure 2.31 – Code used to display favourited food items

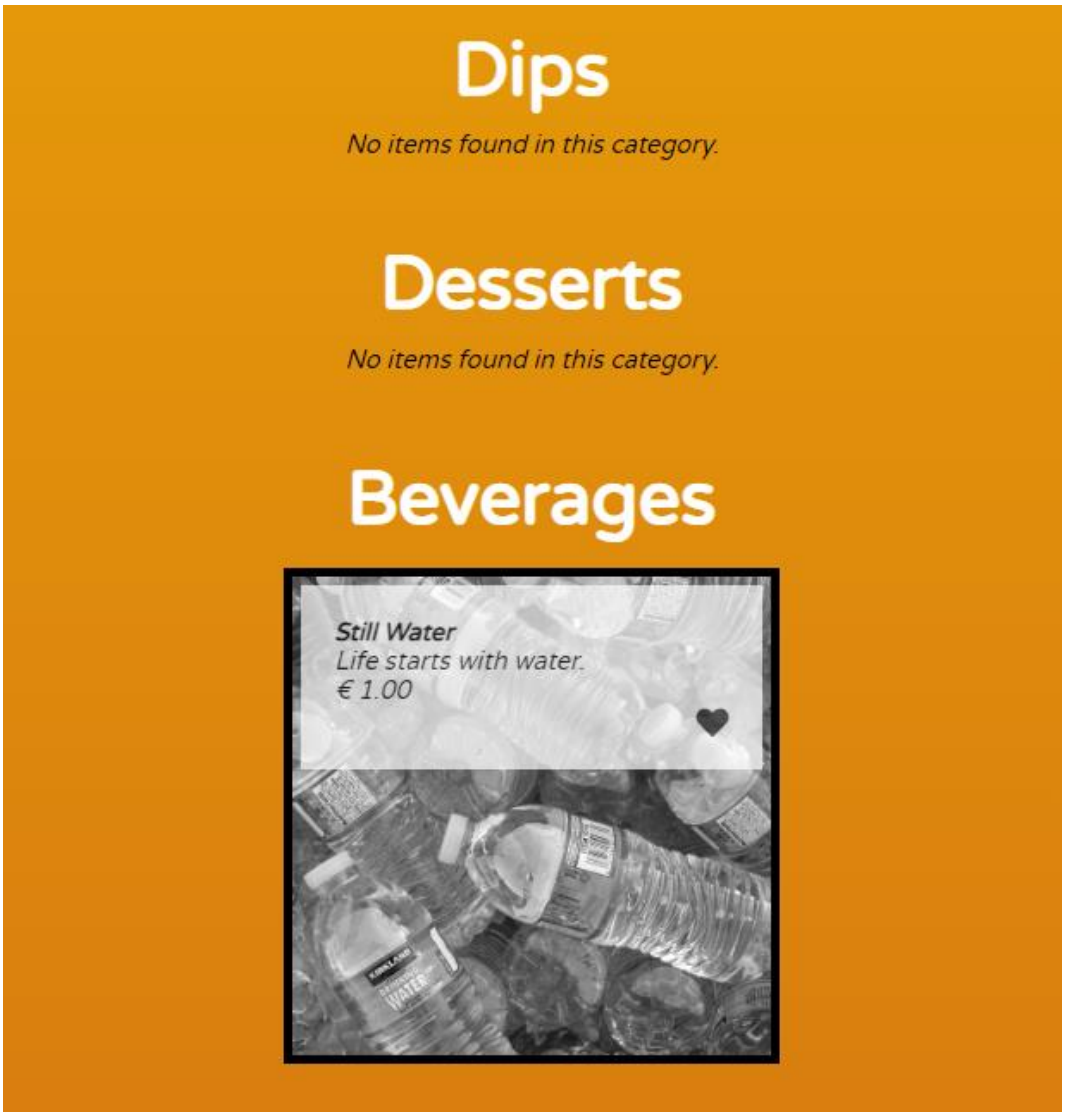


Figure 2.32 – Food Item displayed in the Favourite Page

Following the list of favourited items is the send favourites section. This part allows the user to send the favourites list to a specified email address. When submitted, favouriteResp.php is called to send the email and the appropriate return message is displayed to the user. The above can be seen in Figure 2.33 and Figure 2.34.

```
<!--Allowing the user to send the favourites as an email to a specified address-->
<div class = "favourite-container">
  <form class="email-form" method="post" action="favouriteResp.php">
    <h3 tabindex="0" class = "sec-favourite-title">Send Favourites List to Email</h3>
    <p class ="error">{{ emailError }}{{tnxMessage}}</p>
    <input type="text" id="email" name="email" tabindex="0" placeholder="Email Address" required value="{{ email }}"><br>
    <input class="favourite-input-btn" tabindex="0" type="submit">
  </form>
</div>
```

Figure 2.33 – Code used to display send favourites to email function

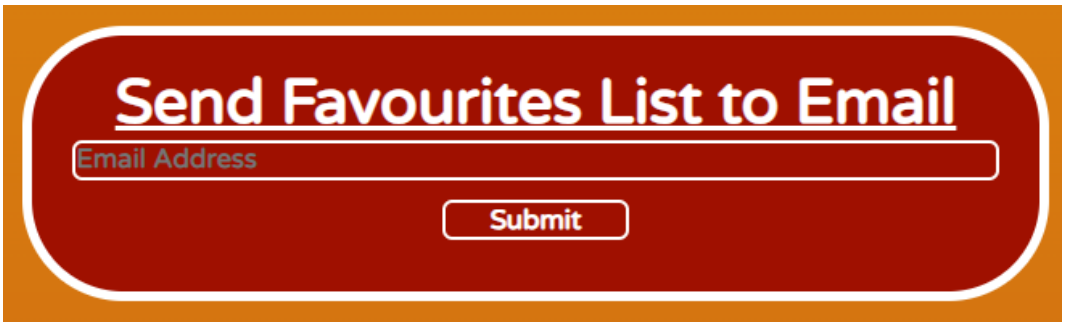


Figure 2.34 – Send favourites list displayed in the Favourite Page

Moreover, the following PHP code was used in conjunction with the favourites.html file to construct the favourites page. TWIG is being used to render the PHP page whilst using the HTML file as a template. The required tables are retrieved from the database. The corresponding parameters are passed as outlined in the layout section. The passed parameters depend on two conditions, whether the get variable 'p' is set and whether it is 0 or 1. If the submit button has not been pressed yet, hence, the variable 'p' is not set, no message is passed to be displayed. However, if it is 0, a thanks message is passed and if it is 1, an error message is passed together with the email inputted. These messages are provided by favouriteResp.php. The above can be seen in Figure 2.35, Figure 2.36 and Figure 2.37.

```php
session_start();
//Acquiring bootstrap.php and database.php
require_once __DIR__.'/bootstrap.php';
require_once __DIR__.'/database.php';

//Loading the necessary code to send an email
include("PhpMailerConfig.php");

//Get the db object
$db = new Db();
//Setting variable check to 0
$check = 0;
//Performing a query to retrieve all the records from the items table
$items = $db -> select("SELECT * FROM items");
//Performing a query to retrieve all the categories from the menu table
$categories = $db -> select("SELECT * FROM menu");
//Setting all variables to empty
$emailErr = $email = $message = "";
```

Figure 2.35 - PHP used in the Favourites Page (Database Retrieval Section)

```php
/*Checking whether it was the first time the user accessed the website
 *If p is set, this would imply that this isn't the first time the user accessed
 *the website*/
if(isset($_GET['p'])){
  //If statement that will execute if the email was sent successfully
  if($_GET['p']==0){
    /*Passing necessary parameters to html template to render the page
     *One of said parameters being a thank you message, since email was sent successfully
     */
    echo $twig->render('favourites.html', array("title" => "Favourites",
    "mastheadImg" => "masthead-favourite",
    "MTitle" => "Favourites List",
    "MSTitle" => "Always a Delight",
    'foodItems' => $items,
    'foodCategories' => $categories,
    'favouritesSession' => $_SESSION['favourites'],
    'check' => $check,
    'tnxMessage' => $_SESSION['tnxMessage']));
  }
  //If statement that will execute if an error occured and email was not sent
  elseif($_GET['p']==1){
    /*Passing necessary parameters to html template to render the page
     *Parameters include the incorrect email entered by the user, as
     *the error which occured.
     */
    echo $twig->render('favourites.html', array("title" => "Favourites",
    "mastheadImg" => "masthead-favourite",
    "MTitle" => "Favourites List",
    "MSTitle" => "Always a Delight",
    'foodItems' => $items,
    'foodCategories' => $categories,
    'favouritesSession' => $_SESSION['favourites'],
    'check' => $check,
    'emailError' => $_SESSION['emailError'],
    'email' => $_SESSION['email']));
  }
}
//Else statement will only execute if it was the first time the user accessed the website
else{
  //Passing necessary parameters to html template to render the page
  echo $twig->render('favourites.html', array("title" => "Favourites",
  "mastheadImg" => "masthead-favourite",
  "MTitle" => "Favourites List",
  "MSTitle" => "Always a Delight",
  'foodItems' => $items,
  'foodCategories' => $categories,
  'favouritesSession' => $_SESSION['favourites'],
  'check' => $check,
  'tnxMessage' => ""));
}
```

Figure 2.36 - PHP used in the Favourites Page (Render Section)

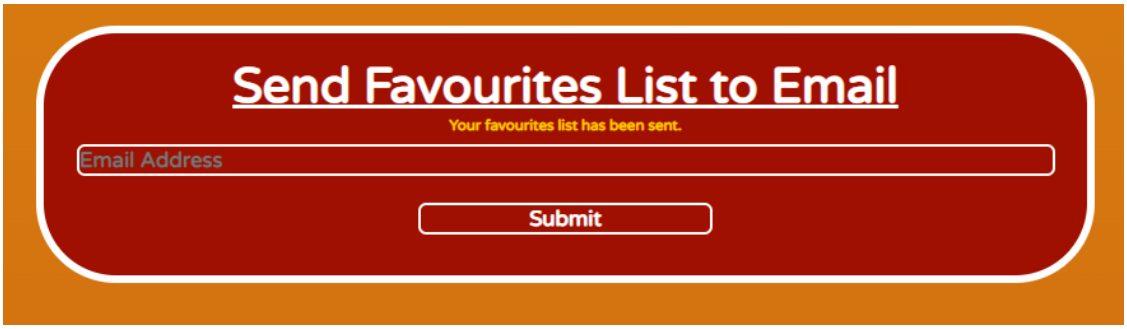


Figure 2.37 - Send favourites list displayed in the Favourite Page with a thanks message

Additionally, the file favouriteResp.php was utilised to construct the email itself. Firstly, similar to favourites.php, the required tables are being retrieved. Following this, the email is constructed by performing error checking on the email address provided and adding the name, description, price, link and image of every favourited food item to the email. If the email address provided is invalid, an error message is passed. Otherwise, a thank you message is passed. The above can be seen in Figure 2.38 and Figure 2.39.

```php
session_start();
//Acquiring bootstrap.php and database.php
require_once __DIR__.'/bootstrap.php';
require_once __DIR__.'/database.php';

//loading the necessary code to send an email
include("PhpMailerConfig.php");

//Get the db object
$db = new Db();
//Setting variable check to 0
$check = 0;
//Performing a query to retrieve all the records from the items table
$items = $db -> select("SELECT * FROM items");
//Performing a query to retrieve all the categories from the menu table
$categories = $db -> select("SELECT * FROM menu");
//Setting all variables to empty
$emailErr = $email = $message = "";
```

Figure 2.38 – PHP used in the favouriteResp file (Database Retrieval Section)

```php
//Check the email field is correct
if(isset($_POST["email"])) {
  if(!empty($_POST["email"])){
    $email = clean_input($_POST["email"]);
    if(!filter_var($email, FILTER_VALIDATE_EMAIL)){
      $emailErr = "*Your email is incorrect*";
    }
    else {
      //Setting the emails which will recieve the message user/owner
      $mail->addAddress($email, '');
      $mail->Subject =  "Favourites List";
      if(is_array($items)){
        foreach($items as $data){
          if (in_array($data['itemId'], $_SESSION['favourites'])) {
            $message .= "Item Name: " . $data['itemName'] . "<br>Item Description: " . $data['itemDescription'] . "<br>Item Price: €" . $data['itemPrice']
            . "<br>More Info: " . "http://localhost/ProjectPart2/web/dishDetails.php?requiredItemId=" . $data['itemId'] . "<br><br>";
            $mail->addAttachment($data['itemImage']);
          }
        }
          //Setting the email message
          $mail->Body = $message;
      }
      //Sending the email
      $mail->send();
    }
  }
  else{
    $emailErr = "*The email field must be filled*";
  }
}

//If no errors have occured redirection occurs and the thankyou message is displayed
if ($emailErr == "") {
  $_SESSION['emailError'] = "";
  $_SESSION['email'] = "";
  $_SESSION['tnxMessage'] = "Your favourites list has been sent.";
  //Redirecting to favourites.php and passing p=0 which is used to denote that no erorrs have occured
  header("location: favourites.php?p=0");
}
else {//If errors have occured redirection occurs and the error message is displayed
  $_SESSION['tnxMessage'] = "";
  $_SESSION['emailError'] = $emailErr;
  $_SESSION['email'] = $email;
  //Redirecting to favourites.php and passing p=1 which is used to denote that an erorr has occured
  header("location: favourites.php?p=1");
}
```

Figure 2.39 – PHP used in the favouriteResp file (Email Section)

Finally, to add or remove an item from the favourites list, the addFavourite.php file is called. The get variables are stored in variables. If the session array favourites is not set, it creates it. If the passed id is not found in the session array, add it to the array. Otherwise, if it is already in the array, remove it. If the get variable 'redirection' was set to 0, redirect the user to the dishDetails.html page. If it is 1, redirect the user to the favourites page. The above can be seen in Figure 2.40.

```php
session_start();
//Saving the variables passed from the get requests
$itemId = $_GET['itemId'];
$redirection = $_GET['redirect'];
$requiredItemId = $_GET['requiredItemId'];

//If the session array favourites is not set, it will set declare it as an array
if(!isset($_SESSION['favourites'])) {
  $_SESSION['favourites'] = array();
}

//Checking if the itemId is not in session array favourites
if (!in_array($itemId, $_SESSION['favourites'])) {
  //Adding item to list
  array_push($_SESSION['favourites'], $itemId);
}//Checking if the itemId is in session array favourites
elseif (in_array($itemId, $_SESSION['favourites'])) {
  //Removing item from list
  $_SESSION['favourites'] = array_values(array_diff($_SESSION['favourites'], array($itemId)));
}

//If original page sources is dishDetails.php
if ($redirection == 0) {
  //Checking if required item is set
  if (isset($requiredItemId)) {
    //Redirecting to dishDetails.php
    header ("location: dishDetails.php?requiredItemId=$requiredItemId");
  } else {
    //Redirecting to menu.php
    header ("location: menu.php");
  }
}//If original page sources is favourites.php
else if ($redirection == 1) {
  header ("location: favourites.php");
}
```

Figure 2.40 – Code used in addFavourites.php file

The email received should be similar to Figure 2.41.

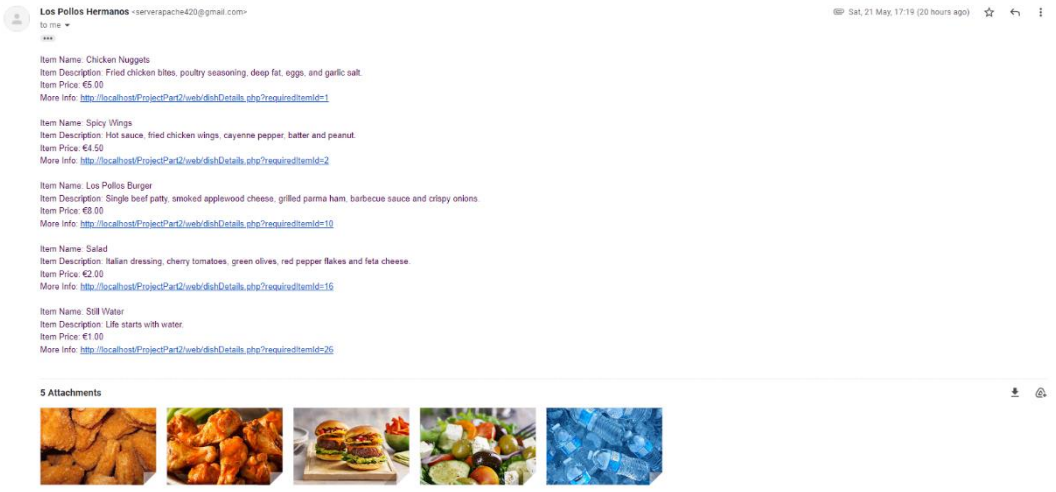


Figure 2.41 – Code used in addFavourites.php file

2.10 Database Setup

2.10.1 General Database Overview

The 'restaurantinfo' database was used to facilitate a multitude of different functionalities required by the website, such as dynamically displaying food items, and many more. The database included the following tables:

- **Items** table holds all the food items which are served at the restaurant
- **Locations** table holds the locations of the restaurant establishments
- **Menu** table holds the different item categories
- **Opening_hours** table holds the opening/closing times for the restaurant
- **Worker_info** table holds information pertaining to the workers

Any changes which are done in the database, will be reflected on the website, as the website is generated dynamically.

The following can all be seen in Figure 2.42.

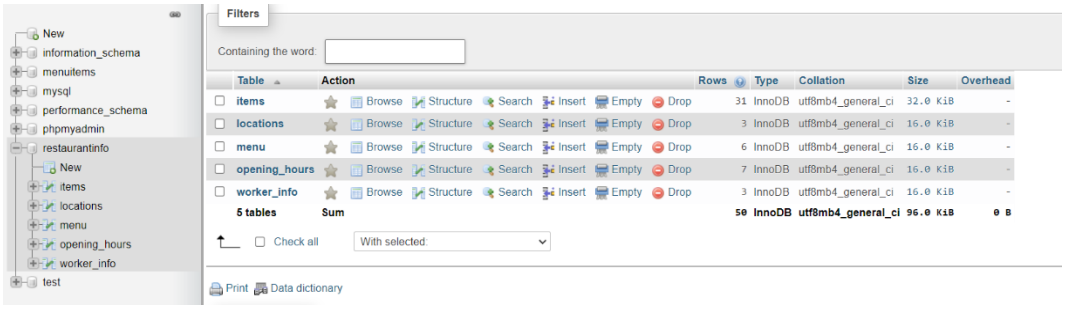


Figure 2.42 – restaurantinfo contents

2.10.2 Items table

The items table schema included the following attributes:

1. itemName holds the name of the food item
2. categoryId is a foreign key which holds the category id from the menu table
3. itemId is the primary key which is used to identify each food item
4. itemDescription holds a brief description/ingredient pertaining to each individual food item
5. itemPrice holds the price pertaining to each food item
6. itemImage holds the path to the image pertaining to each food item

The schema can be seen in Figure 2.43.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	itemName	text	utf8mb4_general_ci		No	None			Change Drop More
2	categoryId	int(11)			No	None			Change Drop More
3	itemId	int(11)			No	None			Change Drop More
4	itemDescription	text	utf8mb4_general_ci		No	None			Change Drop More
5	itemPrice	float(10,2)			No	None			Change Drop More
6	itemImage	text	utf8mb4_general_ci		No	None			Change Drop More

Figure 2.43 – items table schema

Figure 2.44 shows the items table populated with data.

```
SELECT * FROM `items`
```

Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

1 > >> Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

Options

	itemName	categoryId	itemId	itemDescription	itemPrice	itemImage
Edit Copy Delete	Chicken Nuggets	1	1	Fried chicken bites, poultry seasoning, deep fat, ...	5.00	../images/chicken/chicken-nuggets.jpg
Edit Copy Delete	Spicy Wings	1	2	Hot sauce, fried chicken wings, cayenne pepper, ba...	4.50	../images/chicken/spicy-wings.jpg
Edit Copy Delete	Crispy Fillet	1	3	Boneless chicken breasts, buttermilk, eggs, chili ...	4.75	../images/chicken/crispy-fillet.jpg
Edit Copy Delete	Los Pollos Fillet	1	4	Skinless chicken breast, parmesan cheese, olive oi...	5.50	../images/chicken/los-pollos-fillet.jpg
Edit Copy Delete	Los Pollos Meal	1	5	Fried chicken wings, nuggets and a fillet accompan...	10.00	../images/chicken/los-pollos-meal.jpg
Edit Copy Delete	Beef Burger	2	6	Single beef patty, chunky lettuce, caramelized oni...	6.00	../images/burgers/beef-burger.jpg
Edit Copy Delete	Double Beef Burger	2	7	Double beef patties, double cheese, lettuce, tomat...	8.00	../images/burgers/double-beef-burger.jpg
Edit Copy Delete	Chicken Burger	2	8	Fried chicken, bacon, cheese, lettuce, tomatoes an...	6.00	../images/burgers/chicken-burger.jpg
Edit Copy Delete	Spicy Chicken Burger	2	9	Fried chicken, lettuce, caramelized onions, jalape...	7.00	../images/burgers/spicy-chicken-burger.jpg
Edit Copy Delete	Los Pollos Burger	2	10	Single beef patty, smoked applewood cheese, grille...	8.00	../images/burgers/los-pollos-burger.jpg
Edit Copy Delete	Fries	3	11	Raw potato sticks, white vinegar and peanut oil.	1.80	../images/sides/fries.jpg
Edit Copy Delete	Sweet Potatoes	3	12	Sweet potatoes, olive oil, cayenne pepper and sea ...	2.50	../images/sides/sweet-potatoes.jpg
Edit Copy Delete	Onion Rings	3	13	Onions, bread crumbs, egg, baking powder, seasoned...	1.80	../images/sides/onion-rings.jpg
Edit Copy Delete	Mozzarella Sticks	3	14	Mozzarella cheese sticks, bread crumbs, eggs, garl...	0.80	../images/sides/mozzarella-sticks.jpg
Edit Copy Delete	Coleslaw	3	15	Carrots, onions and lettuce mixed lemon juice, vin...	1.90	../images/sides/coleslaw.jpg
Edit Copy Delete	Salad	3	16	Italian dressing, cherry tomatoes, green olives, r...	2.00	../images/sides/salad.jpg
Edit Copy Delete	Mini Kiews	3	17	Ground chicken, eggs, garlic, flour and garlic but...	0.70	../images/sides/mini-kiews.jpg

Console

Figure 2.44 – Records in the items table

2.10.3 Locations table

The locations table schema included the following attributes:

1. store_id is the primary key which is used to identify each location
2. address holds the restaurant addresses

The schema can be seen in Figure 2.45.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	store_id	int(11)			No	None			Change Drop More
2	address	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More

Figure 2.45 – locations table schema

Figure 2.46 shows the locations table populated with data.

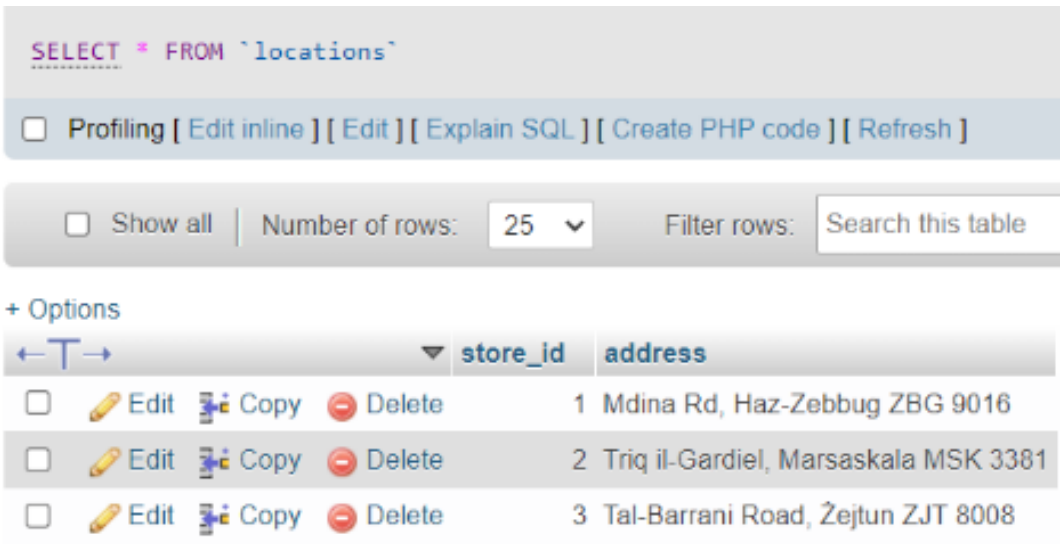


Figure 2.46 – Records in the locations table

2.10.4 Menu table

The menu table schema included the following attributes:

1. categoryID is the primary key which is used to identify each category
2. categoryName holds the different food categories

The schema can be seen in Figure 2.47.

#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	categoryID	int(11)			No	*None*			Change Drop More
2	categoryName	text	utf8mb4_general_ci		No	*None*			Change Drop More

Figure 2.47 – menu table schema

Figure 2.48 shows the menu table populated with data.

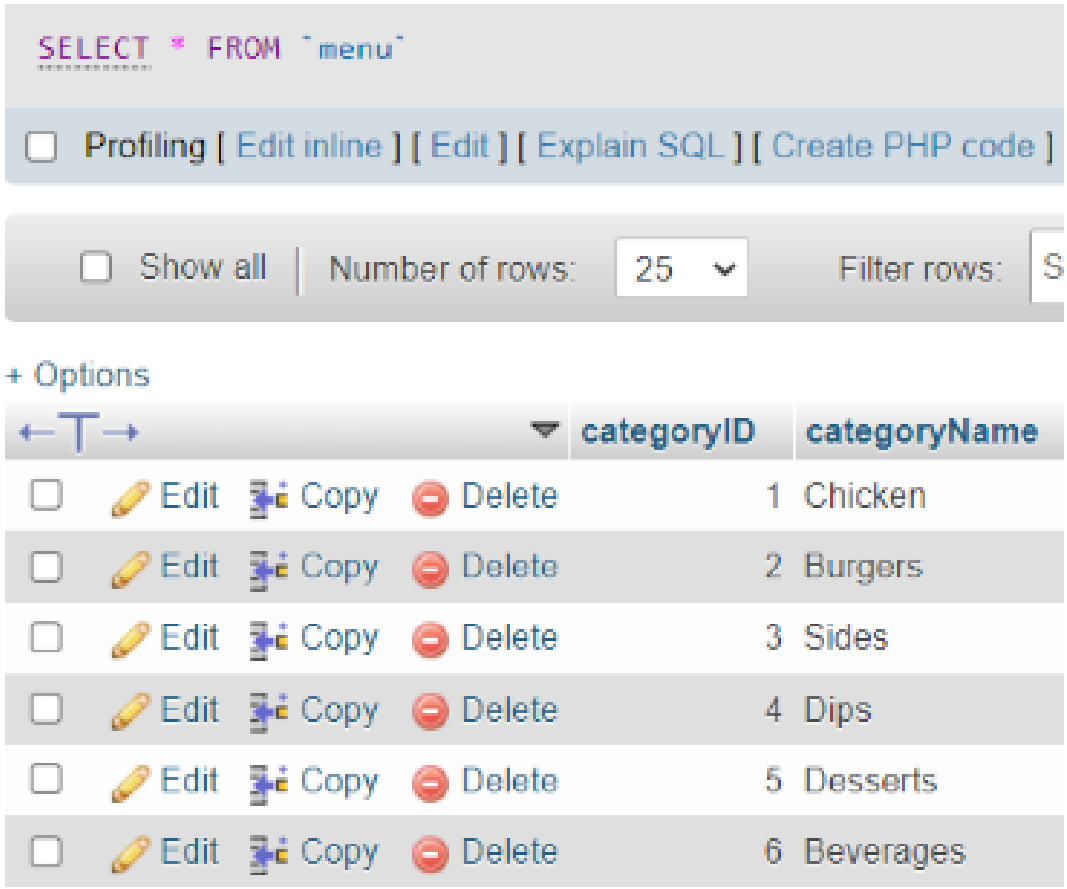


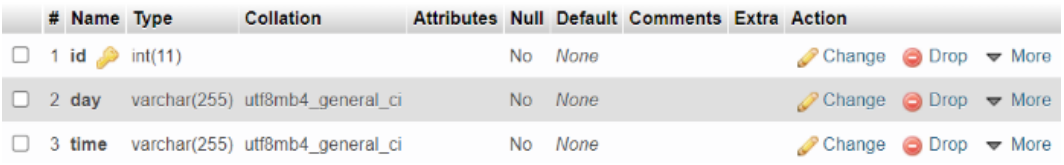
Figure 2.48 – Records in the menu table

2.10.5 Opening_hours table

The opening_hours table schema included the following attributes:

1. id is the primary key which is used to identify each day
2. day holds the current day of the week
3. time holds the time that the restaurant will be open, for each day of the week

The schema can be seen in Figure 2.49.



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	*None*			Change Drop More
2	day	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More
3	time	varchar(255)	utf8mb4_general_ci		No	*None*			Change Drop More

Figure 2.49 – opening_hours table schema

Figure 2.50 shows the opening_hours table populated with data.

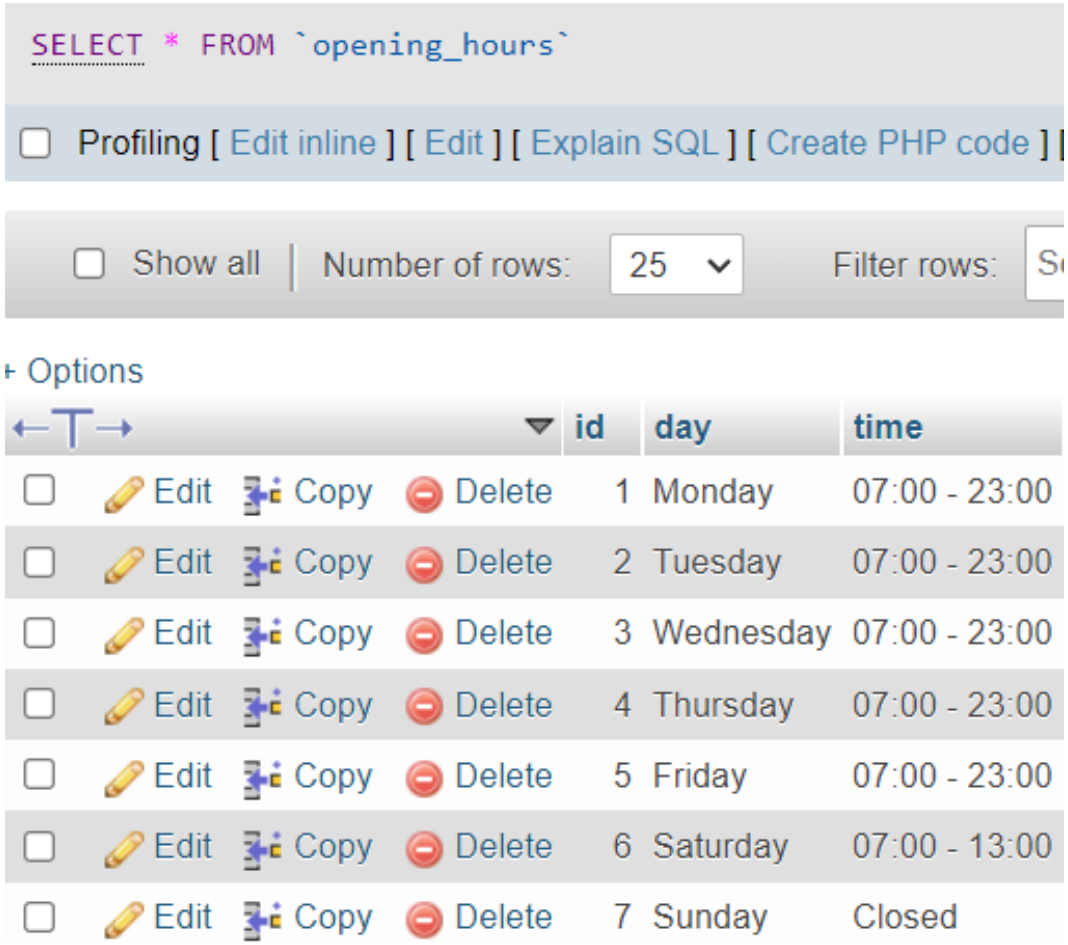


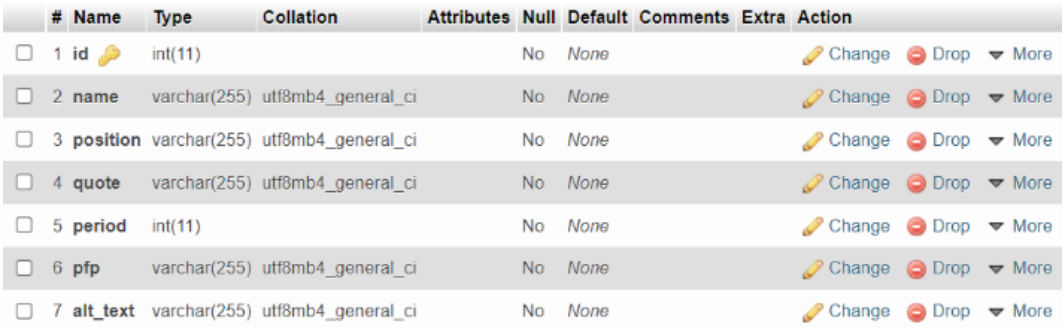
Figure 2.50 – Records in the opening_hours table

2.10.6 Worker_info table

The worker_info table schema included the following attributes:

1. id is the primary key which is used to identify each worker
2. name holds the name of the worker
3. position holds the employee position (manager/owner)
4. quote holds a brief message from each worker
5. period holds the amount of time the worker has worked at the restaurant in years
6. pfp holds the path to the image pertaining to each worker
7. alt_text holds the alternate text which will be used for the pfp image, for html best practices

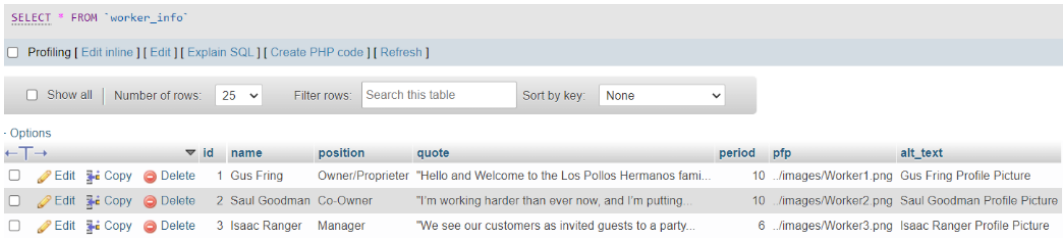
The schema can be seen in Figure 2.51.



#	Name	Type	Collation	Attributes	Null	Default	Comments	Extra	Action
1	id	int(11)			No	None			Change Drop More
2	name	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More
3	position	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More
4	quote	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More
5	period	int(11)			No	None			Change Drop More
6	pfp	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More
7	alt_text	varchar(255)	utf8mb4_general_ci		No	None			Change Drop More

Figure 2.51 – worker_info table schema

Figure 2.52 shows the worker_info table populated with data.



```
SELECT * FROM `worker_info`
```

Profiling [Edit inline] [Edit] [Explain SQL] [Create PHP code] [Refresh]

Show all | Number of rows: 25 Filter rows: Search this table Sort by key: None

Options

	id	name	position	quote	period	pfp	alt_text
Edit Copy Delete	1	Gus Fring	Owner/Proprieter	"Hello and Welcome to the Los Pollos Hermanos fami...	10	../images/Worker1.png	Gus Fring Profile Picture
Edit Copy Delete	2	Saul Goodman	Co-Owner	"I'm working harder than ever now, and I'm putting...	10	../images/Worker2.png	Saul Goodman Profile Picture
Edit Copy Delete	3	Isaac Ranger	Manager	"We see our customers as invited guests to a party...	6	../images/Worker3.png	Isaac Ranger Profile Picture

Figure 2.52 – Records in the worker_info table

2.11 Contact Us Page

The contact.html page consists of the header and footer which were implemented in the layout.html template (described above), as well as two sections, one which will allow the user to book a table, and the other one to send a query/complaint.

The above can be seen in Figure 2.53, Figure 2.54, and Figure 2.55.

```
{% extends "layout.html" %}

{% block content %}
<!--Using extends to avoid repeating code-->

  <!--Displaying Booking form-->
  <section class = "sec-content flexCenter">
    <div class = "booking-container">
      <form class="contact-form" method="post" action="contactResp.php">
        <h3 class = "sec-contact-title">Booking</h3>
        <p class="error">{{ nameErr }}{{ tnxMessage }}</p>
        <input type="text" id="name" name="name" tabindex="0" placeholder="Name" value="{{name}}" required><br>
        <p class="error">{{ emailErr }}</p>
        <input type="email" id="email" name="email" tabindex="0" placeholder="Email Address" value="{{email}}" required><br>
        <p class="error">{{ mobileNoErr }}</p>
        <input type="tel" id="mobile-no" name="mobile-no" tabindex="0" placeholder="Mobile No" value="{{mobileNo}}" required><br>
        <p class="error">{{ noOfPeopleErr }}</p>
        <input type="number" id="no-of-people" name="no-of-people" tabindex="0" placeholder="No of people" value="{{noOfPeople}}" required><br>
        <p class="error">{{ dateErr }}</p>
        <input type="date" id="date" name="date" tabindex="0" placeholder="Date" value="{{date}}" required><br>
        <input class="contact-input-btn" name="booking_btn" tabindex="0" type="submit">
      </form>
    </div>

    <!--Displaying Query/Complaints form-->
    <div class = "contact-container">
      <form class="email-form" method="post" action="contactResp.php">
        <h3 class = "sec-contact-title">Send any Queries/Complaints</h3>
        <p class="error">{{ emailErr2 }}{{ tnxMessage2 }}</p>
        <input type="email" id="email2" name="email2" tabindex="0" placeholder="Email Address" value="{{ email2 }}" required><br>
        <p class="error">{{ subjectErr }}</p>
        <select id="subject" name="subject" tabindex="0" placeholder="Subject" value="{{ subject }}" required>
        <option value="Query">Query</option>
        <option value="Complaint">Complaint</option>
        </select><br>
        <p class="error">{{ messageErr }}</p>
        <textarea class="email-textarea" tabindex="0" name="message" placeholder="Leave a message here!" required>{{ message2 }}</textarea><br>
        <input class="contact-input-btn" name="complaint_btn" tabindex="0" type="submit">
      </form>
    </div>

  </section>
  {% endblock %}
```

Figure 2.53 – Code used to display the two forms

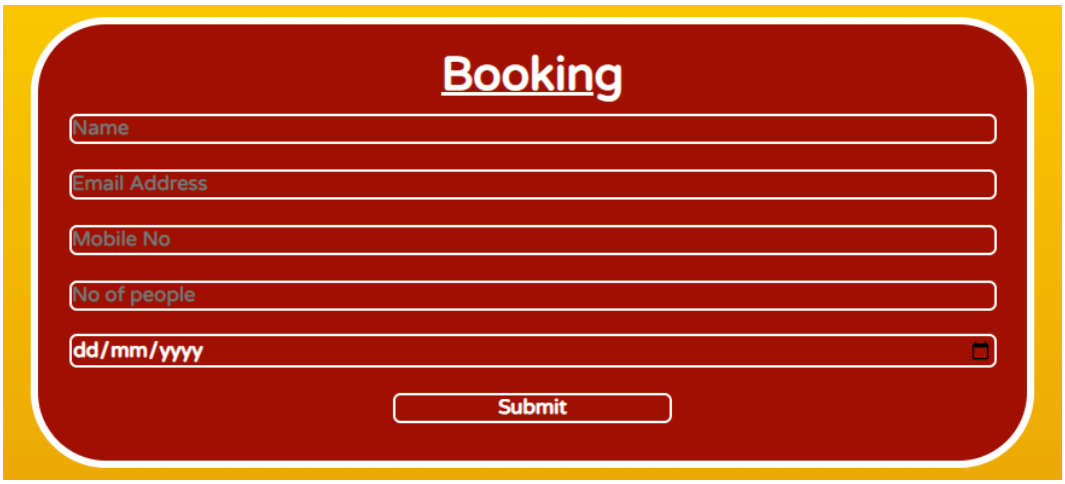


Figure 2.54 – Booking form in Contact Us Page

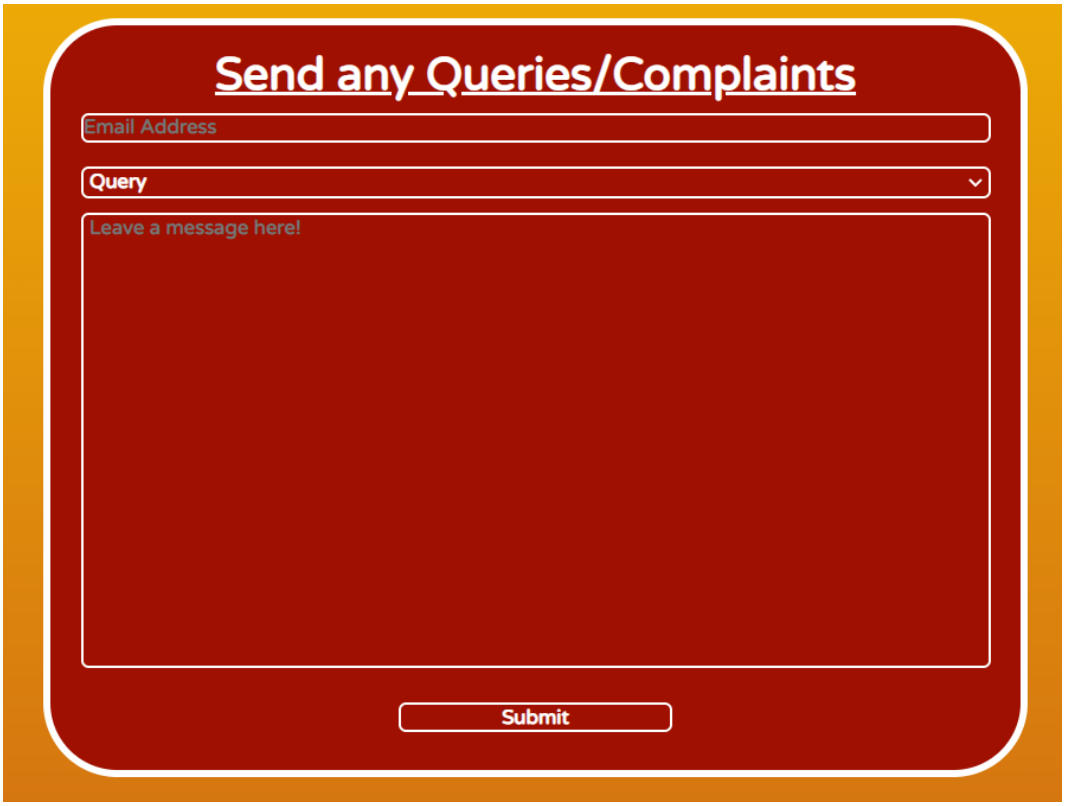


Figure 2.55– Query/Complaints form in Contact Us Page

Moreover, the following PHP code was used in conjunction with the contact.html file to construct the contact page. TWIG is being used to render the PHP page whilst using the HTML file as a template. The corresponding parameters are passed as outlined in the layout section. The passed parameters depend on two conditions, whether the get variable 'p' is set and whether it is 0 or 1. If the submit button has not been pressed yet, hence, the variable 'p' is not set, no message is passed to be displayed. However, if it is 0, a thanks message is passed and if it is 1, respective error messages are passed as well as the email inputted. These messages are provided by contactResp.php.

The above can be seen in Figure 2.56.

```php
<?php
session_start();

require_once __DIR__.'/bootstrap.php';
header("cache-control: private, max-age=86400");

/*Checking if parameter p exists
if yes then the page with the necessaery info is loaded
otherwise the basic page is loaded*/
if(isset($_GET['p'])){
  if($_GET['p']==0){//Loads the page with the required thank you message
    echo $twig->render('contact.html', array(
    "title" => "Contact",
    "mastheadImg" => "masthead-contact",
    "MTitle" => "Contact Us",
    "MSTitle" => "Join us for a tasty meal",
    "tnxMessage" => $_SESSION["tnxMessage"],
    "tnxMessage2" => $_SESSION["tnxMessage2"]));
  }
  elseif($_GET['p']==1){//Loads the page with the specified errors
    echo $twig->render('contact.html', array(
    "title" => "Contact",
    "mastheadImg" => "masthead-contact",
    "MTitle" => "Contact Us",
    "MSTitle" => "Join us for a tasty meal",
    "name" => $_SESSION["name"],
    "mobileNo" => $_SESSION["mobileNo"],
    "noOfPeople" => $_SESSION["noOfPeople"],
    "date" => $_SESSION["date"],
    "message" => $_SESSION["message"],
    "message2" => $_SESSION["message2"],
    "subject" => $_SESSION["subject"],
      "email" => $_SESSION["email"],
      "email2" => $_SESSION["email2"],
      "nameErr" => $_SESSION["nameErr"],
      "mobileNoErr" => $_SESSION["mobileNoErr"],
      "noOfPeopleErr" => $_SESSION["noOfPeopleErr"],
      "dateErr" => $_SESSION["dateErr"],
      "messageErr" => $_SESSION["messageErr"],
      "subjectErr" => $_SESSION["subjectErr"],
      "emailErr" => $_SESSION["emailErr"],
      "emailErr2" => $_SESSION["emailErr2"]));
    }
  }
  else{//Loading the basic version of the page
      echo $twig->render('contact.html', array(
      "title" => "Contact",
      "mastheadImg" => "masthead-contact",
      "MTitle" => "Contact Us",
      "MSTitle" => "Join us for a tasty meal",
      "tnxMessage" => "",
      "tnxMessage2" => ""));
  }
?>
```

Figure 2.56 - PHP used in the Contact Us Page

Additionally, the file contactResp.php is utilised to construct the email itself. The file includes two if statements which are used as error checking for each form. The first if statement checks whether the form attributes are set. This can be seen in Figure 2.57.

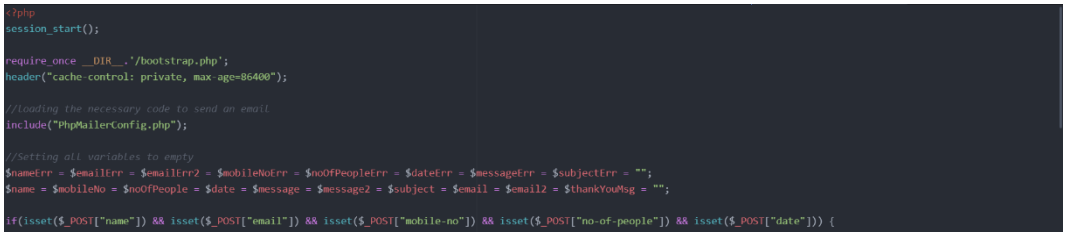


Figure 2.57 – contactResp file header

Following this, each attribute is checked separately for any errors. The name attribute is checked to make sure that it is not empty and fed to the clean_input function. This function will remove any spaces at the beginning and the end of the name attribute, as well as remove any instances of the backslash character and transform any html special characters such that they are ignored by the html interpreter.

Similarly, the email attribute is also checked to make sure it is not empty and fed to the clean_input and filter_val functions. The latter function makes sure that the email has the correct format.

Respective error messages are set which will then be fed to the twig renderer. This can be seen in Figure 2.58.

Figure 2.58 – name and email attributes error checking

The mobile-no attribute is checked to make sure that it is not empty and fed to the clean_input function. Subsequently, the mobile-no attribute is also checked to be 8 characters long.

Similarly, the noOfPeople attribute is also checked to make sure it is not empty and fed to the clean_input function. This attribute is also checked to be in the range of 1-30.

Respective error messages are set which will then be fed to the twig renderer. This can be seen in Figure 2.59.

```
//Check the mobileNo field is correct
if(!empty($_POST["mobile-no"])){
  $mobileNo = clean_input($_POST["mobile-no"]);
  if(strlen($mobileNo) > 8 || strlen($mobileNo) < 8){
    $mobileNoErr = "*Your mobileNo must be 8 characters long*";
  }
}
else{
  $mobileNoErr = "*The mobileNo field must be filled*";
}

//Check the noOfPeople field is correct
if(!empty($_POST["no-of-people"])){
  $noOfPeople = clean_input($_POST["no-of-people"]);
  if($noOfPeople > 30 || $noOfPeople < 0){
    $noOfPeopleErr = "*invlaid number of people per booking*";
  }
}
else{
  $noOfPeopleErr = "*The no of people field must be filled*";
}
```

Figure 2.59 – mobile-no and noOfPeople attributes error checking

The date attribute is checked to make sure that it is not empty and fed to the clean_input and checkdate functions. The latter function makes sure that the email has the correct format. Subsequently the date is also compared to the current date to make sure that the booked date is not in the past.

Respective error messages are set which will then be fed to the twig renderer. This can be seen in Figure 2.60.

```
//Check the date field is correct
if(!empty($_POST["date"])){
  $date = clean_input($_POST["date"]);

  $orderdate = explode('-', $date);
  $month =  intval($orderdate[1]);
  $day   =  intval($orderdate[2]);
  $year  =  intval($orderdate[0]);

  if(checkdate($month,$day,$year) == false){
      $dateErr = "*The date field is invalid*";
  }
  else{
    $now = date('Y/m/d', time());
    if(strtotime($date) < strtotime($now)){
      $dateErr = "*The date field is in the past*";
    }
  }
}
else{
  $dateErr = "*The date field must be filled*";
}
```

Figure 2.60 – date attribute error checking

Finally, the email message is constructed according to the inputted information, and the email subject is set appropriately. If no errors were found, the email and respective thank you message is sent, and the variables are all reset. Furthermore, the user will be redirected to the contact page with p=0 which signifies no error was found. If errors were found, the respective errors are stored in session variables, which will then be accessed and displayed appropriately by the contact page. Furthermore, the user will be redirected to the contact page with p=1 which signifies that the email sending process has failed.

This is reflected in Figure 2.61 and 2.62.

```php
//Setting the email message
$mail->Body = "Name: " . $name . "<br>" . "Mobile No: " . $mobileNo . "<br>" . "No of people: " . $noOfPeople . "<br>" . "Booked Date: " . $date;
//Setting the email subject
$mail->Subject = "Booking";

//If no error were found the email is sent otherwise the necessary error messages are passed to the page to be displayed to the user
if($nameErr == $mobileNoErr && $mobileNoErr == $emailErr && $emailErr == $noOfPeopleErr && $noOfPeopleErr == $dateErr && $dateErr == ""){
  //Sending the email
  $mail->send();
  //Assigning the required thankyou message
  $_SESSION["tnxMessage"] = "Booking received.";
  $_SESSION["tnxMessage2"] = "";
  //Resetting the variable values
  $name = $mobileNo = $noOfPeople = $date = $message = $message2 = $subject = $email = $email2 = "";
}
else{//Passing on the required error messages
  $_SESSION["name"] = $name;
  $_SESSION["mobileNo"] = $mobileNo;
  $_SESSION["noOfPeople"] = $noOfPeople;
  $_SESSION["date"] = $date;
  $_SESSION["message"] = $message;
  $_SESSION["message2"] = $message2;
  $_SESSION["subject"] = $subject;
  $_SESSION["email"] = $email;
  $_SESSION["email2"] = $email2;
  $_SESSION["nameErr"] = $nameErr;
  $_SESSION["mobileNoErr"] = $mobileNoErr;
  $_SESSION["noOfPeopleErr"] = $noOfPeopleErr;
  $_SESSION["dateErr"] = $dateErr;
  $_SESSION["messageErr"] = $messageErr;
  $_SESSION["subjectErr"] = $subjectErr;
  $_SESSION["emailErr"] = $emailErr;
  $_SESSION["emailErr2"] = $emailErr2;
  //Redirecting to contact.php and passing p=1 which is used to denote that an erorr has occured
  header("location: contact.php?p=1");
}
}
```

Figure 2.61 – Construction of email message or storing respective errors in session variables

```php
//If no errors occur the page is redirected to with p=0 signallign no errors and displayign the relevant error message
if($nameErr == $emailErr && $emailErr == $emailErr2 &&
$emailErr2 == $mobileNoErr && $mobileNoErr == $noOfPeopleErr &&
$noOfPeopleErr == $dateErr && $dateErr == $messageErr &&
$messageErr == $subjectErr && $subjectErr == ""){
  //Redirecting to contact.php and passing p=0
  header("location: contact.php?p=0");
}
?>
```

Figure 2.62 – Redirecting to contact.php if there are no errors

The Query/Complaint form was handled similarly concerning error checking.

Figure 2.63 and Figure 2.64 denote the respective error messages which could occur:

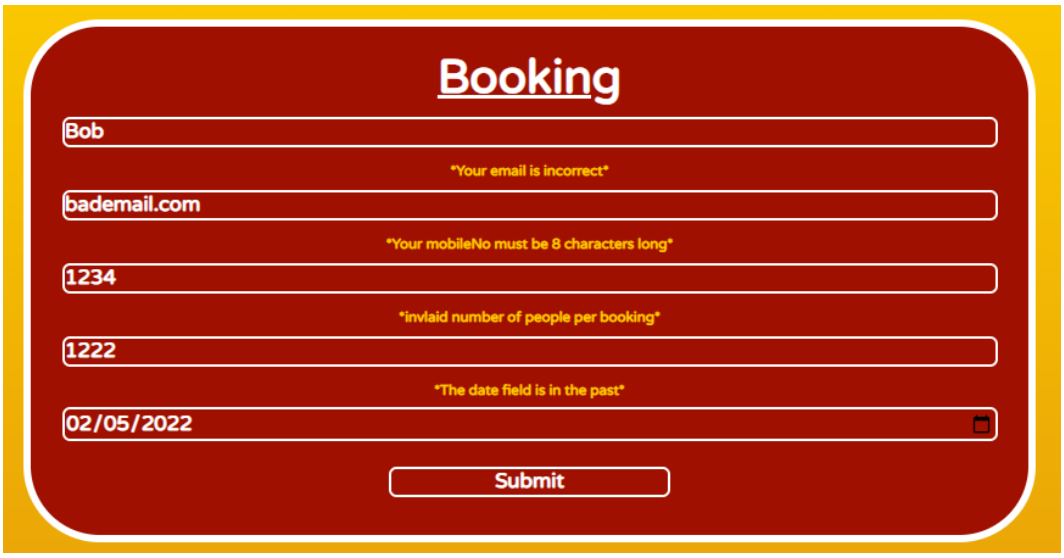


Figure 2.63 – Booking form with errors

Figure 2.64 – Query/Complaint form with errors

The user may opt to choose whether to send a query or complaint, this is facilitated through a drop-down menu. This can be seen in Figure 2.65.

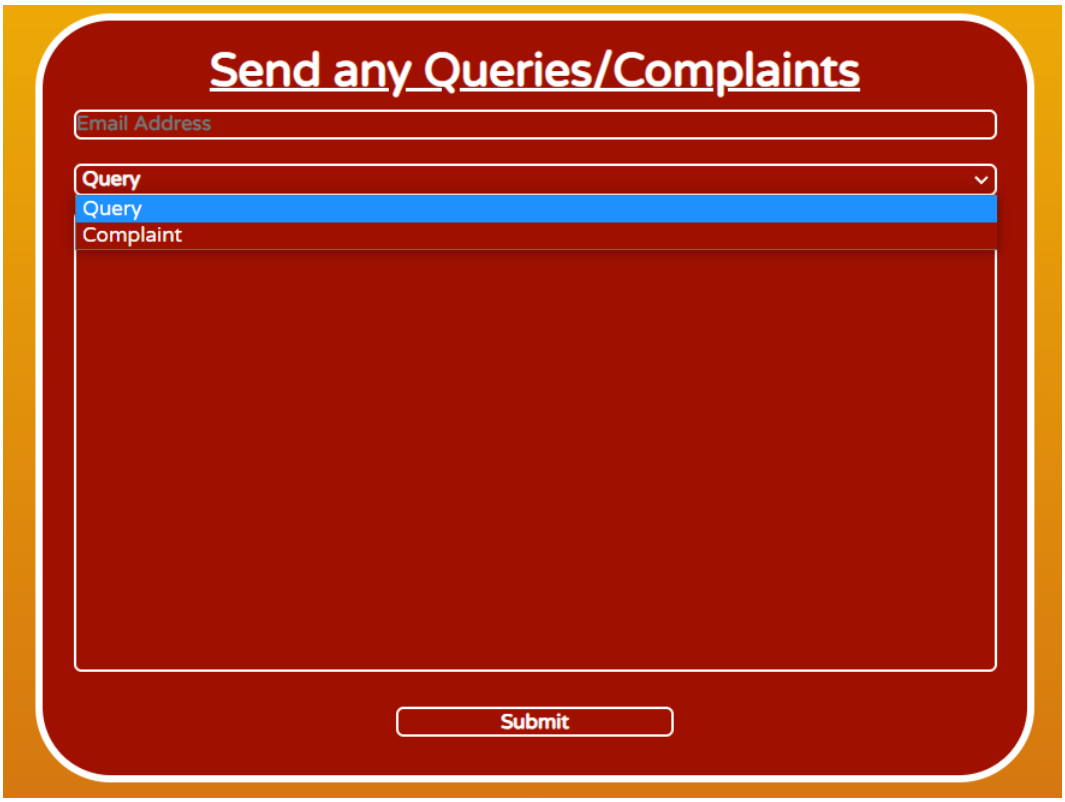


Figure 2.65 – Query Complaint toggle

A thank you message is displayed if the email was sent successfully. This can be seen in Figure 2.66.

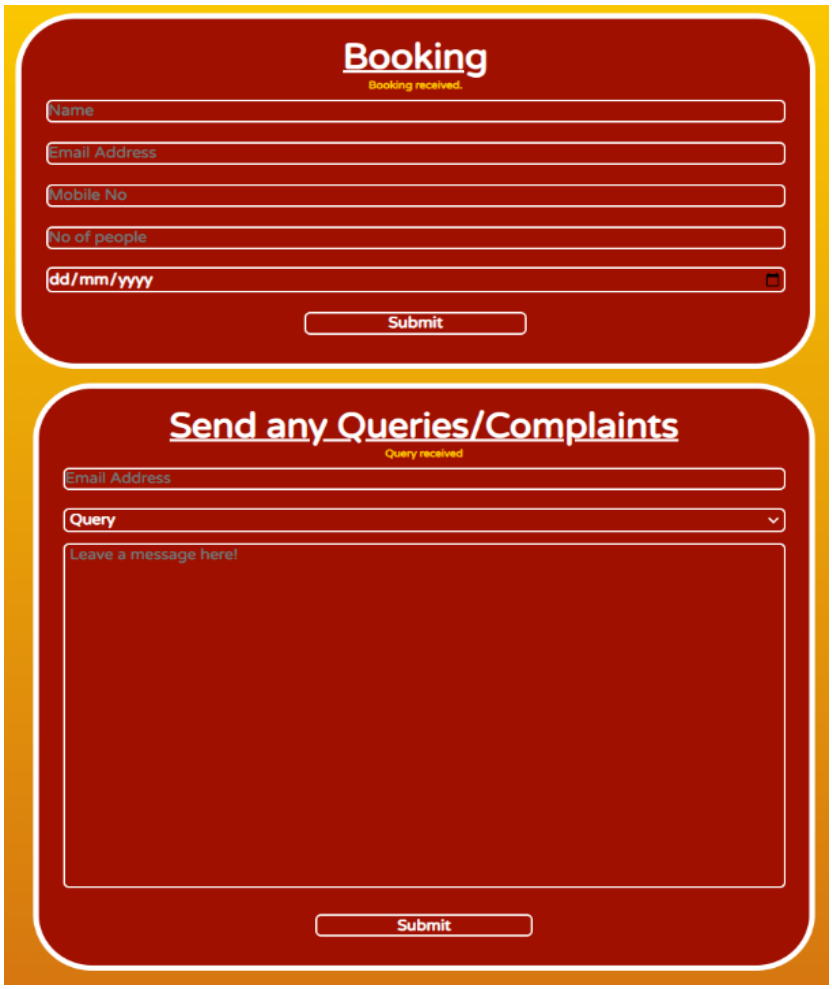


Figure 2.66 – Successful sending of email

Figure 2.67 depicts the emails which will be received by the owner and the user.

Figure 2.67 – Successful sending of email

2.12 Other Files Used

Firstly, PHPMailerConfig.php was used to send emails. The first function called clean_input takes a string of text and performs functions to clean it. The required PHPMailer classes were imported into the global namespace. The Config.ini file is being parsed. A PHPMailer instance was created and was set in such a way that its text is treated as HTML. Debugging was enabled. The hostname, port number, encryption mechanism, authentication method, username, password, sender and reply address were set. Finally, the appropriate SMTP options were selected. The above can be seen in Figure 2.68 and Figure 2.69.

```php
//function used to clean the data and remove any would be harmful characters
function clean_input($data){
  $data = trim($data);
  $data = stripslashes($data);
  $data = htmlspecialchars($data);
  return $data;
}
```

Figure 2.68 – Clean Input Function

```php
//Import PHPMailer classes into the global namespace
use PHPMailer\PHPMailer\PHPMailer;
use PHPMailer\PHPMailer\SMTP;

require 'vendor/autoload.php';

$config = parse_ini_file('Config.ini');

//Create a new PHPMailer instance
$mail = new PHPMailer();

//Tell PHPMailer to use SMTP
$mail->isSMTP();

//Signify that the message sent should be treated as HTML
$mail->isHTML(true);

//Enable SMTP debugging
$mail->SMTPDebug = $config['debug'];

//Set the hostname of the mail server
$mail->Host = $config['smtpServer'];

//Set the SMTP port number:
$mail->Port = $config['port'];

//Set the encryption mechanism to use:
$mail->SMTPSecure = $config['EncryptionMechanism'];

//Whether to use SMTP authentication
$mail->SMTPAuth = $config['DoAuthentication'];

//Username to use for SMTP authentication - use full email address for gmail
$mail->Username = $config['username'];

//Password to use for SMTP authentication
$mail->Password = $config['password'];

//Set who the message is to be sent from
$mail->setFrom($config['fromEmail'], $config['fromAlias']);

//Set an alternative reply-to address
$mail->addReplyTo($config['replyToEmail'],$config['replyToAlias']);

$mail->SMTPOptions = array(
        'ssl' => array(
                'verify_peer' => $config['verify_peer'],
                'verify_peer_name' => $config['verify_peer_name'],
                'allow_self_signed' => $config['allow_self_signed']
            )
         );
```

Figure 2.69 – Setting appropriate mail settings

The file database.php contains multiple functions related to accessing a specified database. The connect function connects to a database and returns the database upon success. The query function returns the result of a query in the database. The select function returns the records specified from a table in a database. The error function returns the most recent error that occurred. The quote function returns a quoted string word. The above can be seen in Figure 2.70.

```php
class Db {
    // The database connection
    protected static $connection;

     //Try connect, returning false on failure
    public function connect() {

        //Try connecting to the database
        if(!isset(self::$connection)) {
            // Load config file into array
            $config = parse_ini_file('./Config.ini');
            self::$connection = new mysqli($config['server'], $config['dbUsername'], $config['dbPassword'], $config['dbName']);
        }

        //If connection was not successful, return false
        //If successful, return connection
        if(self::$connection === false) {
            return false;
        }
        return self::$connection;
    }

    // Query the database
    public function query($query) {
        //Connecting to the database
        $connection = $this -> connect();

        //Querying the database
        $result = $connection -> query($query);

        return $result;
    }

    //Fetch records from table
    public function select($query) {
        $rows = array();
        $result = $this -> query($query);
        if($result === false) {
            return false;
        }
        while ($row = $result -> fetch_assoc()) {
            $rows[] = $row;
        }
        return $rows;
    }

    //Fetch the last error
    public function error() {
        $connection = $this -> connect();
        return $connection -> error;
    }

    //Quote and escape value for use in a database query
    public function quote($value) {
        $connection = $this -> connect();
        return "'" . $connection -> real_escape_string($value) . "'";
    }
}
```

Figure 2.70 – Code in database.php

The file Config.ini contains any specified information regarding database access or SMTP settings. These variables are accessed in PHPMailerConfig.php and database.php. Note that emails are sent from serverapache420@gmail.com while pollosowner@gmail.com receives a copy of the email. The above can be seen in Figure 2.71.

```
[database]
dbUsername = 'RestaurantUser'
dbPassword = 'JerMatIsa'
dbName = 'restaurantinfo'
server = 'localhost'

[smtp]
port = 587
debug = 0
smtpServer = 'smtp.gmail.com'
username = 'serverapache420@gmail.com'
password = 'JerMatIsa'
fromEmail = 'serverapache420@gmail.com'
fromAlias = 'Los Pollos Hermanos'
replyToEmail = 'pollosowner@gmail.com'
replyToAlias = 'Los Pollos Owner'

verify_peer = false
verify_peer_name = false
allow_self_signed = true

EncryptionMechanism = 'PHPMailer::ENCRYPTION_STARTTLS;'
DoAuthentication = 'true'
```

Figure 2.71 – Config.ini Content

The .htaccess file specifies the time cached content can be utilised before being updated again. Also, access to certain pages is forbidden and error pages are set depending on the type of error. The above can be seen in Figure 2.72.

```
Options -Indexes

# enable the directives for this application
ExpiresActive on

# send an Expires: header for each of these mimetypes
ExpiresByType image/png "access plus 1 month"
ExpiresByType image/gif "access plus 1 month"
ExpiresByType image/jpeg "access plus 1 month"

# css may change more frequently, so a shorter expiration period is recommended
ExpiresByType text/css "access plus 1 days"

# Setting a custom 404 page
ErrorDocument 404 /ProjectPart2/web/error.php
ErrorDocument 403 /ProjectPart2/web/error.php

<FilesMatch "\.(ini|htaccess)$">
  Deny from all
</FilesMatch>
```

Figure 2.72 – .htaccess Contents

2.13 Best Practices

1. Separating PHP from HTML

Using Twig each webpage was separated into two files one which contained html code and the other PHP. This leads to the webpage logic being separated from the markup, which will make the code more readable. This is apparent in Figure 2.73.

```
//Passing necessary parameters to html template to render the page
echo $twig->render('main.html', array("title" => "Main Page",
"mastheadImg" => "masthead-main",
"MTitle" => "Welcome to Los Pollos Hermanos",
"MSTitle" => "It's finger lickin good",
"locations" => $locations,
"openingHours" => $openingHours));
```

Figure 2.73 – Making use of the render function

2. JavaScript, PHP, HTML, CSS Best Practices

CSS and JavaScript scripts were split into different folders. This is useful as it avoids having inline CSS & JavaScript scattered amidst the HTML/PHP. This can be seen in Figure 2.74.

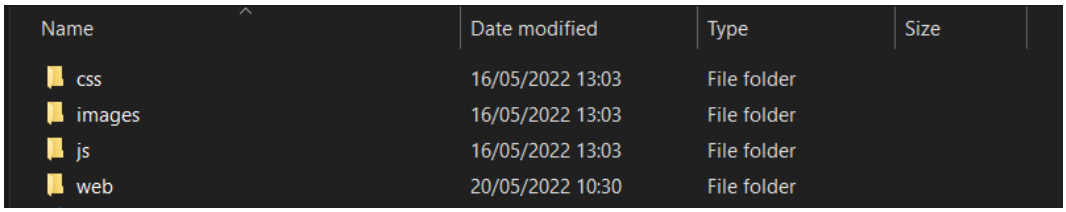


Figure 2.74 – Functionality separated in folders

3. Construction of Dynamic Web pages, and retrieving data from Database

The home, menu, favourites, and workers pages are all loaded dynamically by retrieving their required data from a database. This lends itself to having the webpage be easily updatable by the unqualified user, from the database. This is apparent in Figure 2.75.

```
<!--Looping through every food item Row -->
{% for food in foodItems %}
  <!--Checking if the food item categoryId corresponds to the Categories' categoryID-->
    {% if food.categoryId == Categories.categoryID %}
      <!--Displaying relevant attributes from food item Row-->
      <tr class = "table-data italics">
        <!--The food item name, will serve as a link to the dishDetails.php page, with the item's Id as a parameter, when clicked-->
        <td><a class="menu-link" tabindex="0" href="dishDetails.php?requiredItemId={{ food.itemId }}">{{ food.itemName }}</a></td>
        <td class = "align-right"><a tabindex="0"> € {{ food.itemPrice }}</a></td>
      </tr>
    {% endif %}
  {% endfor %}
  </tbody>
</table>
{% endfor %}
```

Figure 2.75 – Menu items being dynamically generated

4. Server and User Side Error Checking for email sending

The favourites & contact pages make use of both server side & user side validation. This is done so that the user cannot inject any code into the system and if the user has JavaScript disabled the appropriate error checking is still done. This can be seen in Figure 2.76.

```
//Check the name field is correct
if(!empty($_POST["name"])){
  $name = clean_input($_POST["name"]);
}
else{
  $nameErr = "*The name field must be filled*";
}

//Check the email field is correct
if(!empty($_POST["email"])){
  $email = clean_input($_POST["email"]);
  if(!filter_var($email, FILTER_VALIDATE_EMAIL)){
    $emailErr = "*Your email is incorrect*";
  }
  else{
    //Setting the emails which will recieve the message user/owner
    $mail->addAddress($email, '');
    $mail->addAddress($config['replyToEmail'], $config['replyToAlias']);
  }
}
else{
  $emailErr = "*The email field must be filled*";
}
```

Figure 2.76 – Error checking functionality

5. Screen Reader/Tab Functionality

All the webpages are disability friendly as they support traversal via the tab button as well as working appropriate with screen readers. This was achieved through the tab index and title properties. This is apparent in Figure 2.77.

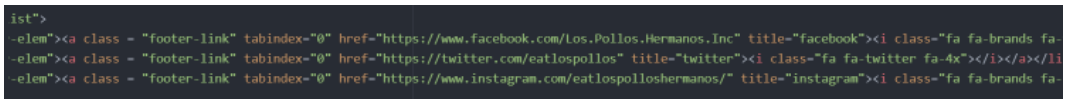


Figure 2.77 – Using tabindex and title properties

6. DRY Principles

To avoid code duplication, two layout pages were used, one for the general page and the other for the error pages. The header.html and footer.html files were also used to display the header/footer and the PhpMailerConfig.php file was used to hold the settings in relation to sending emails. This can be seen in Figure 2.78.

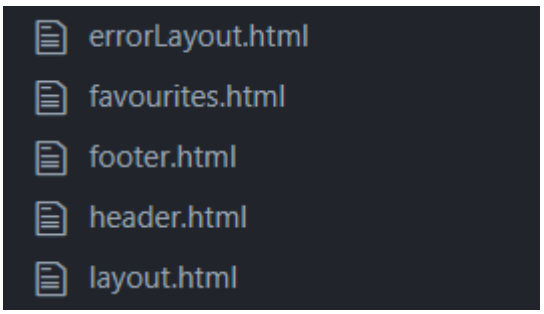


Figure 2.78 – Showing layout pages

7. Custom Error pages

The website is designed to handle errors by displaying an appropriate error page in relation to 404/403 errors, which provides a link to the user to go back to the main page. This can be seen in Figure 2.79.

Figure 2.79 – Custom Error Page

8. Cache Control

Each webpage has its associated cache control settings which signals to the browser to keep the pages cached for a maximum of 1 day. This will reduce server traffic, but the website wouldn't be out of date as it will update every day. This is apparent in Figure 2.80.

```
//Setting Cache Control
header("cache-control: public, max-age=86400");
```

Figure 2.80 – Showing Cache Control setting

9. Site Responsiveness

Each page makes use of the appropriate media queries to style the page to fit different screen sizes. This can be seen in Figure 2.81.

```
@media (max-width: 788px) {
  .info-img, .info-par{
    width: 50%;
    height: auto;
  }
```

Figure 2.81 – Applying a media query

10. Using a config.ini file

A config file is used in relation to the sending emails and accessing the database, this can easily be altered by the website administrator to alter any of these settings if required. This is apparent in Figure 2.82.

```
[smtp]
port = 587
debug = 0
smtpServer = 'smtp.gmail.com'
username = 'serverapache420@gmail.com'
password = 'JerMatIsa'
fromEmail = 'serverapache420@gmail.com'
fromAlias = 'Los Pollos Hermanos'
replyToEmail = 'pollosowner@gmail.com'
replyToAlias = 'Los Pollos Owner'

verify_peer = false
verify_peer_name = false
allow_self_signed = true

EncryptionMechanism = 'PHPMailer::ENCRYPTION_STARTTLS;'
DoAuthentication = 'true'
```

Figure 2.82 – Config.ini file

11. Database limited user access

The database user has limited access to avoid catastrophic problems in case of database illicit access. This can be seen in Figure 2.83.

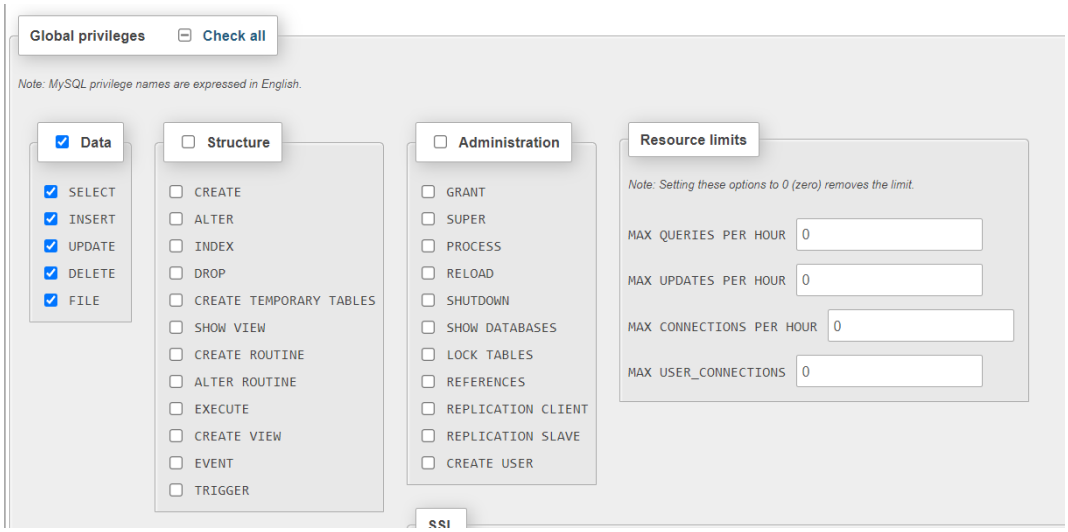


Figure 2.83 – RestaurantUser Privileges to access database

12. Alternate text to be displayed for images

All images have alt text which is displayed in case the image doesn't load. This is done so that the user can still have an idea of what the webpage included. This is apparent in Figure 2.84.

```
<img tabindex="0" class = "info-img" src = "../images/generic-img1.jpg" alt = "Los Pollos Founders Together" title = "Los Pollos Founders Tog
<p tabindex="0" class = info-par>Located in every corner of the island, Los Pollos Hermanos takes pride in its mouthwatering meals.
  With a menu ranging from the juiciest burgers to crispy fried chicken wings, we provide some of the best meals in the country.
  The name, translating to "The Chicken Brothers", truly represents what our restaurant stands for, eating chicken with family and friends.
  Los Pollos Hermanos has 4 total restaurants in Malta and 1 in Gozo. These are found in San Giljan, Mellieha, Haz-Zebbug, Rabat and Marsalfo
  Founded in 2012 by the partners, we specialise in all types of chicken. Wings? Nuggets? Fillets? You name it and we've got it!</p>
```

Figure 2.84 – Making use of alt text

13. Site Interactions

The links/clickable webpage sections are all styled in such a way so as to signal to the user the presence of extra functionality. This can be seen in Figure 2.85.

Figure 2.85 – Showing hover effects

14. Form Best Practices

All forms present in the website make use of redirection techniques to avoid resending the form on page reload. This is apparent Figure 2.86.

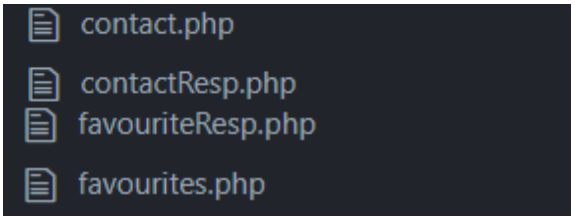


Figure 2.86 – Resp sites used for redirection

3. Part 3 – Distribution of Work

We all had equal parts in the project such that we all did a similar amount of work on all of the project's components.

Github Members:
Matthias Bartolo – mbar0075
Jerome Agius – Jer0me123
Isaac Muscat – isaac21muscat

The project was built across a span of multiple online calls. These are the following dates:

08/04/2022 – Setting up Git Repository and Adding Team Members
12/04/2022 to **14/04/2022** – Uploaded/Worked on Project Part 1 Draft
18/04/2022 – Fixed Certain Errors, General Code Optimisation and Uploaded Updated Files
15/05/2022 – Worked on and Added question1.php
19/05/2022 – Updated README file and Fixed Spelling Mistakes
21/05/2022 – Worked on Section 1 Documentation
26/05/2022 – Final Checking

12/04/2022 – Setting up Git Repository and Adding Team Members
14/04/2022 to **15/04/2022** – Uploaded/Worked on Main Page
16/04/2022 to **17/04/2022** – Uploaded/Worked on Mobile Nav Bar, Error Page, Workers Page, Updated CSS (Added Media Queries), Added Images and Started Building the Database
19/04/2022 – Attempted to Add Database Files
21/04/2022 - **24/04/2022** – Uploaded/Worked on contact.php, Implemented Email Functionality, Added tabIndex, the Code in mobileNav.php and its respective CSS was implemented to the other Web Pages and the temporary file was removed
25/04/2022 – Uploaded/Worked on menu.php and menuGet.php
26/04/2022 to **01/05/2022** – Modified menu.php, Uploaded/Worked on addFavourite.php / Favourites.php / dishDetails.php, Improved CSS implementation in php files
06/05/2022 – Updated CSS / General Changes

12/05/2022 to **13/05/2022** – Twig and Server-Side changes, the header and footer common to each page was moved to two separate php files which were then included into each page via an include statement

14/05/2022 to **19/05/2022** – Added .htaccess, databaseConfig.ini and phpMailer.ini, Redid email implementation to make use of PHPMailer instead of Mercury, added contactResp.php and favouriteResp. Other pages were overhauled to make use of TWIG renderer. The appropriate template pages were added as well as the general page layout.

21/05/2022 to **23/05/2022** - Worked on Section 2 Documentation

23/05/2022 – Uploaded Finalised Project

24/05/2022 – Added README file to Git

26/05/2022 – Updated README file, Updated .htaccess, Updated Documentation, Final Checking