INSTITUTO NEBRIJA | Formación Profesional

ANTONIO DE NEBRIJA 500 AÑOS

Grado Superior en Desarrollo de Aplicaciones Multiplataforma

José Manuel Montaño Mengual
Proyecto Integrador
1er Curso

Versión	Fecha	Motivo de modificación	Elaboración	Revisión	Aprobación
1.0.0	14.06.24	Entrega	José Manuel Montaño Mengual		

Proyecto Integrador

1º Curso DAM

INSTITUTO NEBRIJA | Formación Profesional

ANTONIO DE NEBRIJA 500 AÑOS

i. Abstract

This is the documented memory for the end of course project of the 1st year of the Cross-platform Application Development associate degree of the Nebrija Institute. The project is a private e-commerce platform for a chef, where he can manage his menu, receive and manage orders as well as manage user roles. Logged in users can access their orders or place some.

This project is a prototype, meaning that it is not intended for production, but rather for applying core programming concepts such as layered design, MVC architecture and much more that will be explained throughout this document. The project uses vanilla PHP 8.3.6 and MySQL 8.0.36 to achieve a fully functional full-stack web application. I also created a router framework for vanilla PHP inspired by Gio[1] which I will later proceed to explain.

ii. Acknoledgments

To my dear friend Carlos,

without whom,

I would not have discovered my passion for programming.

iii. Demo

Source code: https://github.com/josemontano1996/nebrija-proyecto-integrador-1

The webpage has been deployed to a VPS in clouding.io. A LAMP stack was installed and configured via command line using SSH credentials. This demo will be only available for the examination period due to cost reasons. If after this examination period you are interested in seeing the webpage contact me at jm3develop@gmail.com and I will reactivate the server for you. Only main pages, meaning home, menu, cart and login pages are fully responsive.

The demo data is the following:

IP address: http://161.22.40.113/

Demo users: there are 3 demo users to try all the features of the webpage, **please do not change any of the demo account information or delete any of the demo pre populated data** as this can affect people watching the page after your.
I recommend that, **if you want to update or delete something from the menu**, to add a new dish and then update it or do whatever you want with it for testing purposes.
If you want to change user roles or change an account data, simply create a dummy account, it takes 2 seconds, promote it to the role you want from the owner account and then play with it all you want, but please leave all the demo data as it is. If by mistake you break something please contact me. All of the accounts can change their account data under the Account section.

- Owner: there is one owner privileged account, as owner it can manage the menu and user roles.
    - email: owner@owner.com password: proyecto-2024

- Admin: there is one admin privileged account, as admin it can manage the menu.
    - email: admin@admin.com password: proyecto-2024

- User: there is one user account, as a user you can create and manage your orders.
    - email: user@user.com password: proyecto-2024

1. Introduction

It is not surprising that the internet revolutionized and is revolutionizing the way companies reach their customers and generate sales. With the growth of internet usage, being online is every time more important as customers are getting more used to the online shopping experience. However, many small businesses struggle to maintain a modern website due to the ever-evolving nature of web development. This can lead to lost sales, decreased customer engagement, and a competitive disadvantage.

Of course, this is not completely the business fault, web development has suffered tremendous changes in the last 15 years: the rise of mobile-first development, PHP upgrades up to version 8, plus all the improvement in its frameworks, and what I think it has been to most important revolution in the last 15 years in web development, the whole rise of JavaScript based frameworks based on the V8 Google Engine, that completely broke the paradigm and created blazingly fast UIs giving a better user experience and raised expectations of what users demand from a modern web application.

Considering it is difficult for small businesses to keep up with the revolutionary speed at which the web development industry evolved I came out with the idea of this project, and although you may see it as a simple e-commerce platform, the intent of this project is to focus on the food and beverage sector, check which common points do food delivery businesses have, and implement a prototype that can be used in order to implement and modernize the online services of restaurants and other food providers , which share a common core, extensible through the whole industry, making it easier, faster, securer and cheaper to maintain than other alternatives.

In this prototype I mocked the case of Chef Manuelle, a private chef that wants to sell his meals online and needs a webpage to do so.

1.1 Study case

In order to make this project, I studied the core necessities of what a private chef needs to receive orders that adapt to certain needed requirements. To achieve this and better understand the needs of this kind of business I analyzed pages like ubereats.com or lieferando.de , I also interviewed some local chefs to retrieve as much vital information about the core functionalities and security checks that the web application should have.

This security check includes ensuring that the prices set in the menu items are updated and that the user does not manipulate them while creating the order. Another example is the implementation of

minimum required quantities, with which, the specific elements of the menu that require a minimum quantity are only purchased when this quantity is reached.

1.2 General description

The web app is designed to be implemented in any server that can run PHP 8.1 or higher, and MySQL 8.0, making the hosting possibilities available countless, in this case I will deploy it to a private VPS, installing a LAMP stack via command line.

The structure of the web app is composed of a public side (front-end) and a private side (back-end). The web app utilizes a custom router framework, which stores the registered routes and resolves the incoming requests to give the user (if allowed) the resources they demand. The general web flow of a request is the following:

- The user makes a request to the server.
- All incoming requests that are not public files are filtered via an entry point (index.php) thanks to the .htaccess file configuration.
- In the entry point the request is completely validated, and if all checks are passed the route gets resolved through the custom Router class.
- The router framework calls the respective controller classes and methods and then the requests follow the path through the layers of the MVC controller until it sends back the response.

The web application has 4 main layers and use cases:

- Public layer: where visitors can navigate through non protected routes.
- User layer: where logged in users can manage their data, create and manage their orders.
- Admin layer: where the administrative personnel can perform tasks such as managing the menu, visualizing and managing orders.
- Owner layer: where the owners of the business can do all the tasks that are included in the admin layer, plus user management.

2. Analysis

2.1 Main objectives

The main objective of this project is the implementation of a web app where the study case, Chef Manuelle, can manage the menu, receive orders and manage them as well as manage the users.

The main objectives of this web application are:

Infrastructure layer

- The web app must be accessible via the internet.
- The web app must be secure and protect data integrity.
- The key parts of the web app must adapt to mobile devices.

Administrative layer

- The Chef or personnel with owner privileges must be able to manage other users' roles.
- The administration personnel must be able to create a menu and completely manage it, creating, editing, or deleting elements of it.
- The administration personnel must be able to visualize and manage orders.

Users layer

- All visitors can see the menu created by the Chef.
- Visitors can create an account to become users of the app.
- Logged in users can place orders and manage them.
- Logged in users can manage their account data.

2.2 Secondary objectives

- The web app must help improve conversion rates of the business owning the webpage.
- The web app must help improve the brand image.

2.3 Requirements

2.3.1 Functional requirements

- RF1 The app should be available 99.9% of the time.
- RF2 The app should be reachable from anywhere via the internet.
- RF3 The app must be secure and protect all sensible information.
- RF4 The app must be responsive and adapted to mobile devices.
- RF5 The app must manage resources in the most efficient way possible.

2.3.2 Specific requirements

- Public side
  - REV1 Visitors can navigate through non protected areas.
  - REV2 Visitors can create an account to become users of the app.

- Private layer (users)
  - REU1 Registered users can manage their account data.
  - REU2 Registered users can create orders.
  - REU3 Registered users can manage their orders.

- Private layer (admin)
  - REA1 Admins can manage the menu.
  - REA2 Admins can manage orders.

- Private layer (owner)
  - REO1 Has access to all admin functionalities.
  - REO2 Can manage user privileges.

2.3.3 System requirements

- RS1 Computer with internet connection.
- RS2 LAMP stack consisting of:
  - Linux OS, Apache server, PHP 8.1 or higher and MySQL 8.0.

2.4 Use cases diagrams

An use case diagram is a type of UML diagram that visually represents the interactions between a system and its users. It is focused on the functionalities (use cases) that the system provides and showcases how different user types (actors) interact with them in order to make the system functional.

The main components of a use case diagram are:

- Actors: Those are the external entities, for example a web visitor, that will interact with the system. They are showcased as human stick figures.
- Use cases: Those represent the functionalities offered by the system. It is represented as the encircled use case.
- System: It represents the system itself, in our case our web app. It is represented by a rectangle that encloses the use cases.
- Relationships: They represent the interaction of the use cases with the actors. They are represented by different sorts of lines depicting association relationships, inclusion and exclusion relationships amongst others.

Some of the benefits that come from the usage of use case diagrams in the analysis stage of software development are the following:

- Improves understanding of the core functionalities of the project.
- Helps identify and document the system's functionalities.
- Clearly defines the boundaries of the system as well as who will have access to each functionality.

In our web app we can distinguish 4 actors:

- Not logged in users:
    - Visitor: every person that visits the web app and is not logged in.
- Logged in users:
    - User: an User is a person that has no role in our database, and as such has no increased privileges.
    - Admin: Admins are users with special permissions, their role is saved as 'admin' in the database.
    - Owner: Owners are the users with most privileges, their role is saved as 'owner' in the database.

2.4.1 Visitor

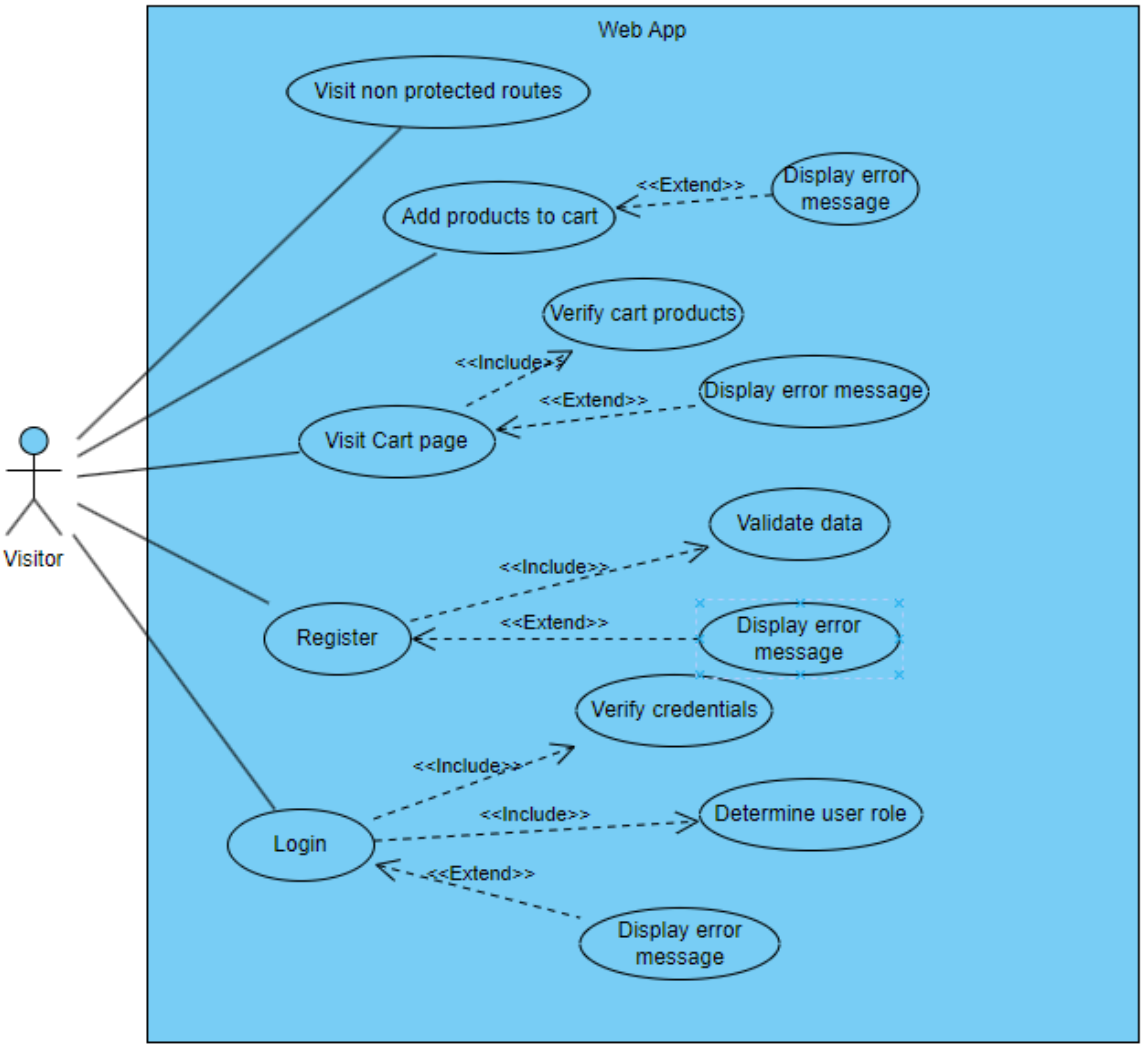


Diagram 1. Visitor use case diagram

2.4.2 User

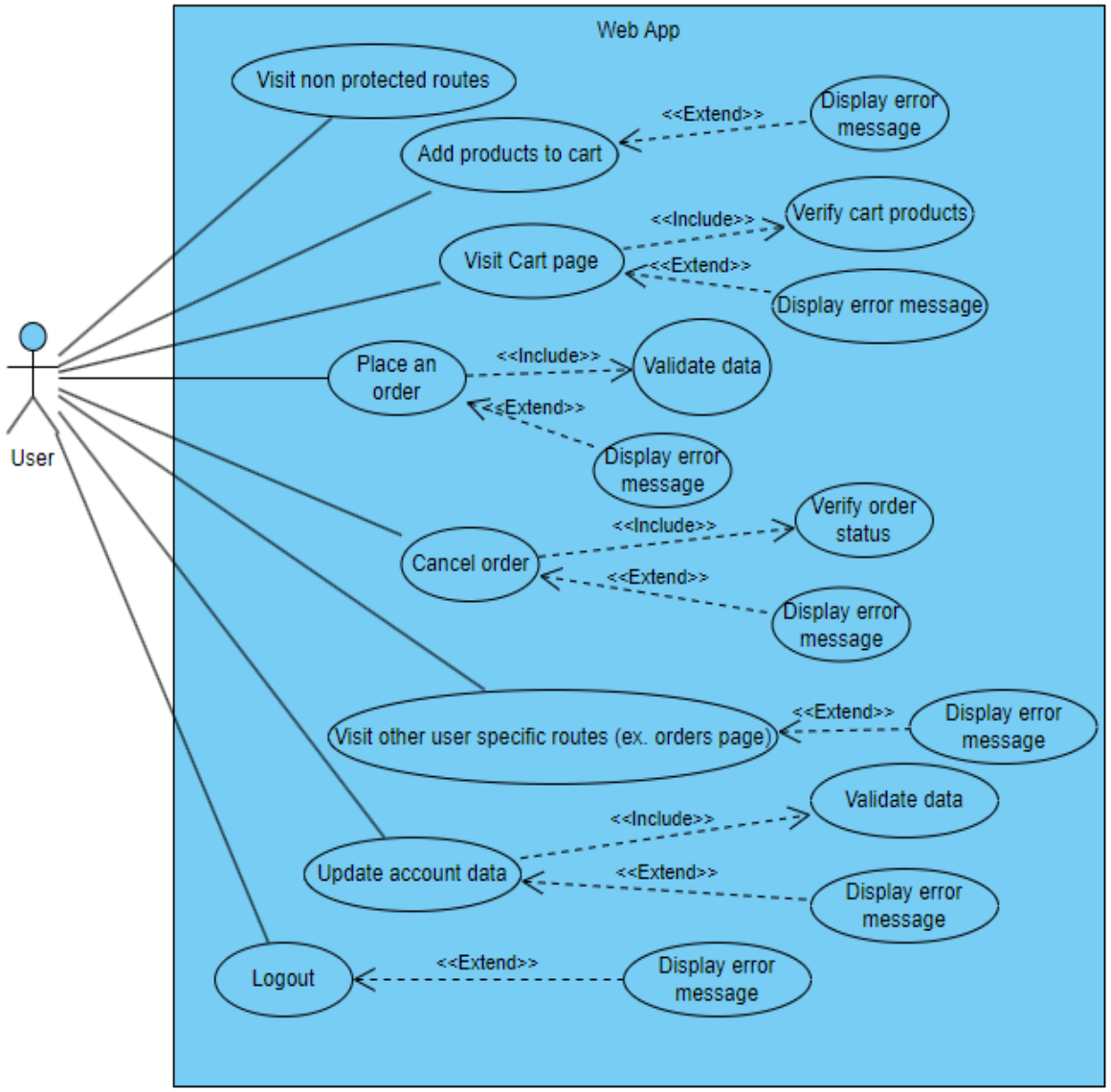


Diagram 2. User use case diagram

2.4.3 Admin

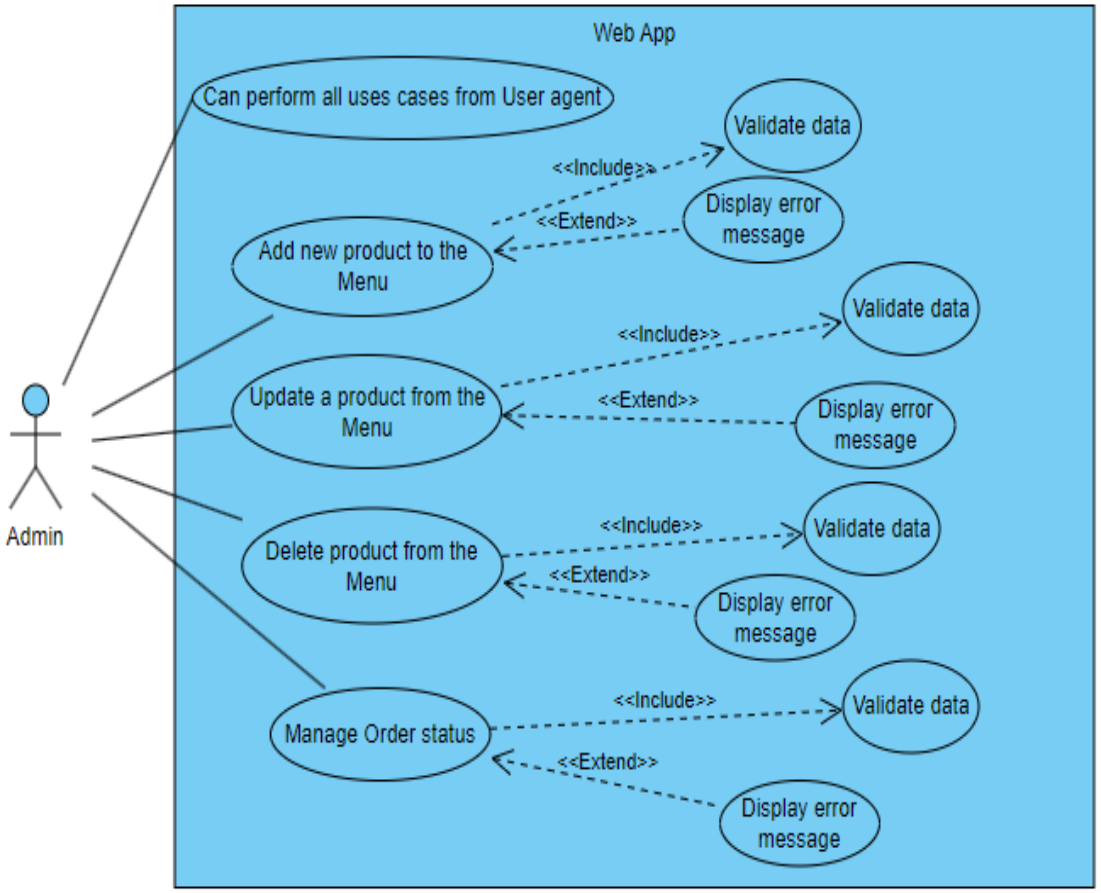


Diagram 3. Admin use case diagram

2.4.4 Owner

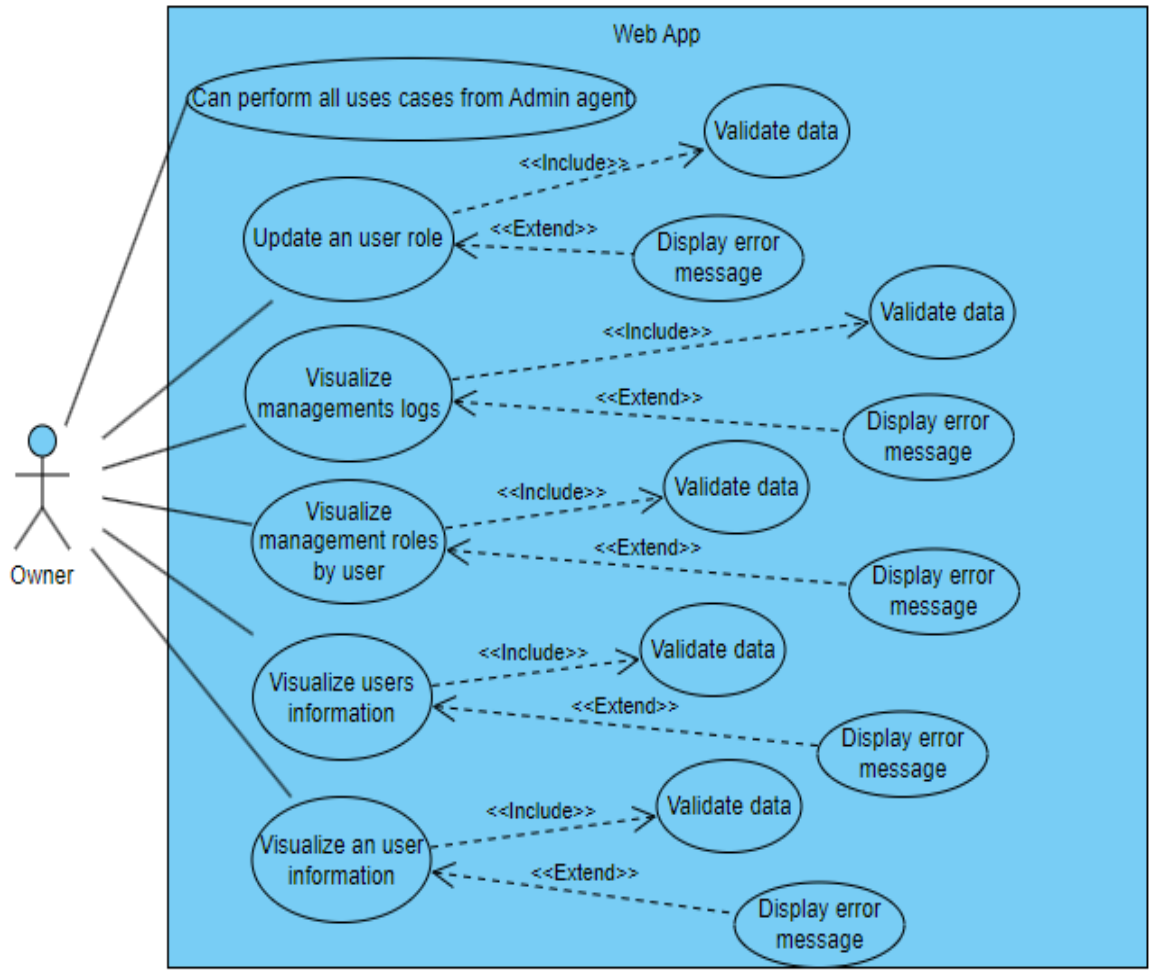


Diagram 4. Owner use case diagram

3. Design

3.1 Programming paradigm

A programming paradigm is a fundamental approach towards writing computer programs. It consists of a set of principles, methods, and practices that dictate how code should be structured, organized and executed.

For this project we have mainly followed the Object Oriented Programming paradigm, which we will discuss in the next section.

3.1.1 Object Oriented Programming

Object Oriented Programming, also known as OOP, is a programming paradigm that revolves around objects. Objects are self-contained reproducible units that bundle data (attributes) as well as related behavior (methods) that operate that data. This makes the code pretty maintainable and predictable, making it pretty suitable for medium to large scale projects.

There are 4 core principles that every OOP program should follow and that dictate how objects are designed and how they interact.

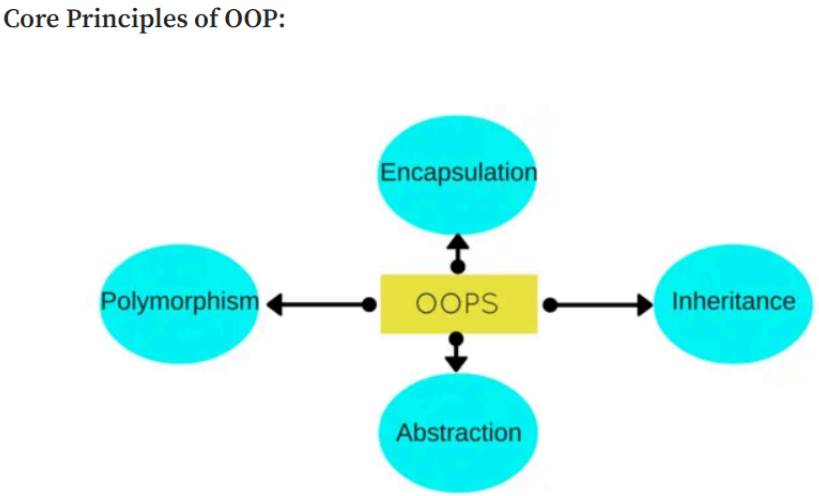


Image 1. Core Principles of OOP [2]

1. *Encapsulation*: encapsulation is the bundling of data (attributes) and behavior (methods) within a single unit called object. Thanks to access modifiers, such as, *public*, *private* and *protected* we can control the access to the different attributes and methods of the object, thus restricting access to the data and providing more control over data management, avoiding unwanted behavior.

2. *Abstraction*: abstraction is the process of simplifying complex systems by focusing on the vital aspects while disregarding irrelevant details. This is achieved with the use of types, interfaces, methods and properties that showcase what the object will do, but hides the actual needy greedy details about how this is accomplished. This can be furthermore enhanced by using some design patterns such as Dependency Injection, about which we will discuss later in the Design Patterns section in page 25.

3. *Inheritance*: inheritance is the mechanism by which a class (subclass) can inherit properties and methods from other classes (superclass). Subclasses can extend or override inherited properties or methods. This enables code reusability, reducing code duplication and promoting code maintainability and extensibility.

4. *Polymorphism*: polymorphism is the mechanism by which objects use a common set of methods or functions to interact with objects of different classes, treating them as if they all belong to a shared superclass. Polymorphism allows the same method to behave differently depending on the object that invokes it.

   Polymorphism can be achieved by different ways such as:

   a. Interface implementation: Interfaces define the signature of an object, setting which methods that class is implementing as well as the expected arguments and outputs of them. Different classes can implement the same interface allowing the classes to be treated polymorphically thanks to the shared interface.
   b. Method overriding: this occurs in the context of inheritance, when a subclass provides a specific implementation of a method defined in its superclass. When the method is called in the subclass the overridden method is being invoked.
   c. Method overloading: although not specific to polymorphism, it can be used to achieve it. In this case polymorphism occurs when a class has a set of methods with the same name but different parameters list. The correct method is called depending on the number and type of arguments given.

3.2 Architectural design

The architectural design in software development defines the high-level structure of the software system, settling the core pillars upon which the software would be builded upon. It defines the major components, their functionalities and purpose, as well as how they interact with each other in order to get the result we want to achieve in an standardized, understandable and structured manner.

3.2.1 Client-Server architecture

The client-server architecture is an architectural model that divides the computer network into two types of computers: the client and the server.

- Client: those are computers or devices (laptops, smartphones, tablets) that request services and that use client programs, such as web browsers to make a request to a server.
- Server: a server is a computer program whose function is to respond/serve to the request made by the client. The servers provide functionality and serve the clients to fulfill the requests. A single server can serve multiple clients at the same time and there are multiple types of servers, for example, web servers (Apache, Nginx…) that serve HTTP requests, database servers (MySQL, MariaDB, MongoDB…) that run Database Management Systems in order to store information and serve it to the client and much more.

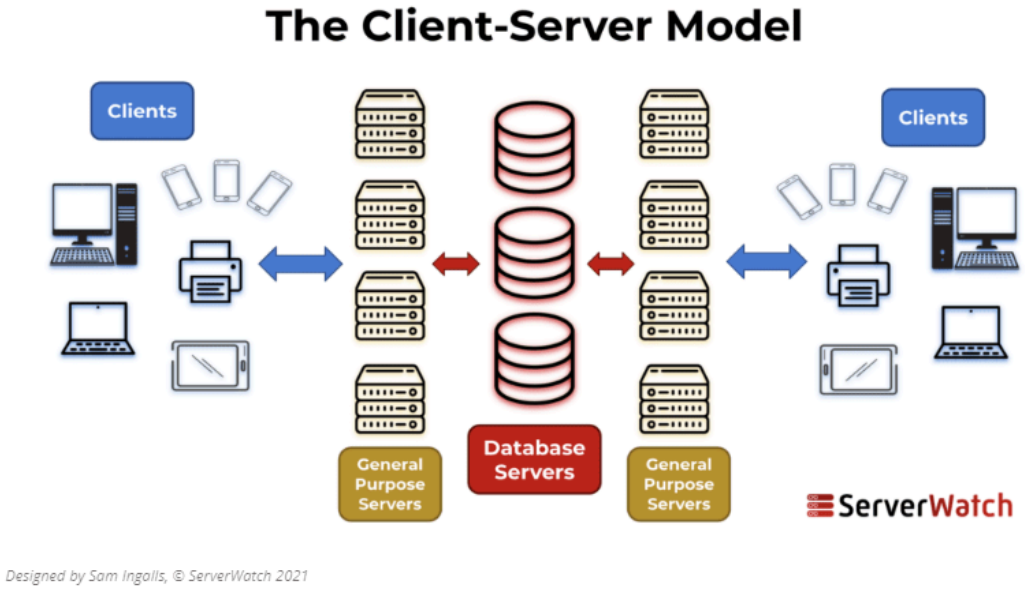


Image 2. The Client-Server Model [3]

This communication between both computers is achieved by a request-response cycle, where the client sends a request to the server and this one sends back a response to that request. This communication is achieved on the web via the HTTP protocol or HTTPs protocol.

HTTP and HTTPS are TCP/IP based protocols that allow web applications to communicate and exchange data in a standardized manner. The key difference lies in security. HTTPS uses TLS (Transport Layer Security) to encrypt the communication between the browser and the web server. This encryption scrambles the data, making it more difficult to decipher to anyone intercepting it in the middle. This makes HTTPS much more secure for transmitting sensitive information like login credentials or financial data.

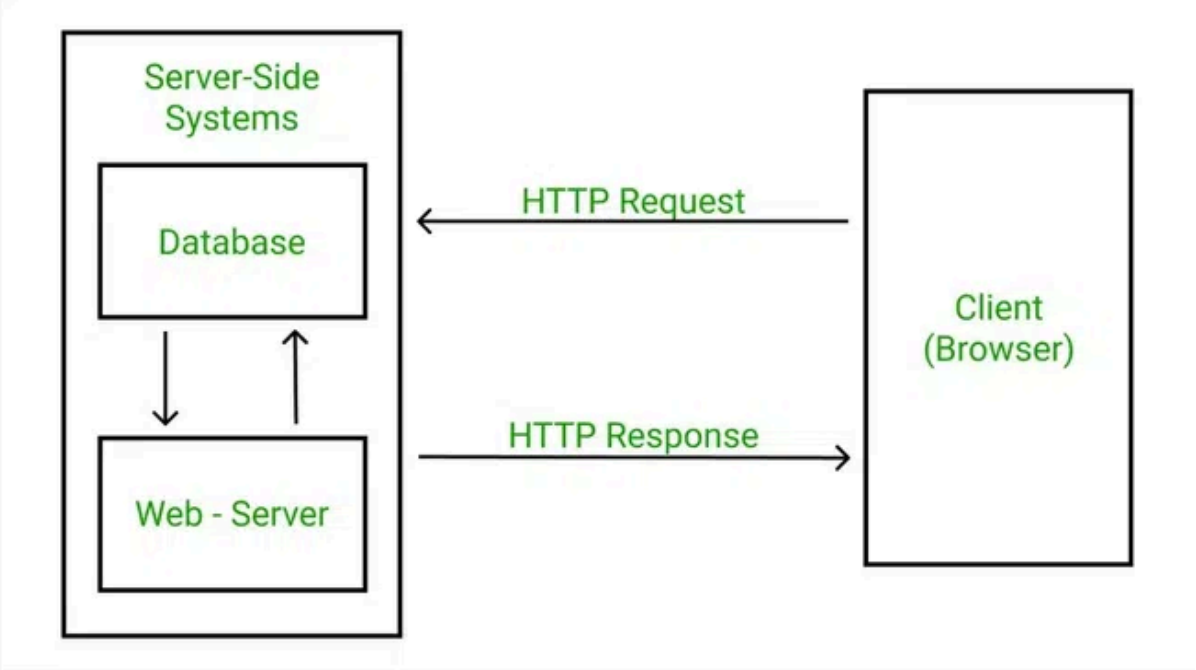


Image 3. The Client-Server Model [4]

The HTTP request-response cycle follows the following flow:

1. Client initiates a request.
    1.1. The user interacts with the client program (usually a browser) by typing an URL or clicking in a link.
    1.2. The browser performs a DNS (Domain Name System) lookup in order to resolve the given URL into a numerical IP address where the requested resources are located.
2. Client builds the HTTP request.

    The HTTP request contains:

    2.1. Headers: they provide additional data like the type of data being requested, request

method or cookies.

2.2. The message body: this only applies to some methods like for example POST and it contains the actual data sent in the request, for example the data written in a form.

3. Client sends the request through TCP/IP to the designated server port.
4. Server processes the request.

   This processing step can consist of:

   4.1. Executing business logic.
   4.2. Accessing and performing database operations.
   4.3. Interacting with other services.
5. Server builds the HTTP response.

   The HTTP response contains:

   5.1.1. Status: a numerical code that indicates the outcome of the request, for example 200 for success, 404 for not found and so on.
   5.1.2. Headers: they can contain information about the response content type or cookies that need to be sent back to the client.
   5.1.3. Message body: it contains the actual data requested by the client, for example, the HTML content of a web page.
6. Server sends the HTTP response to the client through the TCP connection.
7. Client receives and processes the response.

Which benefits bring the HTTP protocol?

1. Standardization: HTTP defines a common language for clients and servers which allows any web browser to interact with any web server that adheres to HTTP protocol.
2. Simple and flexible: the request-response cycle is pretty comprehensive and straightforward which makes it easy to understand and implement.
3. Stateless communication: each HTTP request-response cycle is independent, so the server does not need to care about remembering previous interactions with the client, making it pretty scalable for concurrent requests.
4. Cacheable content: HTTP allows content to be cached by the clients and intermediary servers, improving website loading times where the data is directly retrieved from a cache rather than the requesting them again from the server.

3.2.2 Layered architecture

The layered architecture is an architectural pattern that structures an application into a set of layers, each being responsible for a specific function. This is pretty helpful in promoting key concepts such as:

- Separation of concerns: each unit of the system focuses on a particular aspect of the system, without worrying about the rest of the system. This is pretty helpful when structuring software, because we can exactly identify which parts of our systems do what, helping maintainability, clarity and scalability.
- Modularity: the system is broken into independent self-contained modules within each layer, making the code easier to understand, maintain and modify.
- Directed dependencies: in a well organized layered system the dependencies transpire unidirectionally. Upper layers rely on services provided by lower layers, making the flow of our system pretty well defined.

For this project we have used 4 layers which are the following:

1. Presentation layer: this layer handles user interaction, such as displaying information or managing user interactions.
2. Business logic layer: in this layer the core business functionalities are implemented.
3. Data Access layer: manages interactions with the database.
4. Database layer: stores the application's data.

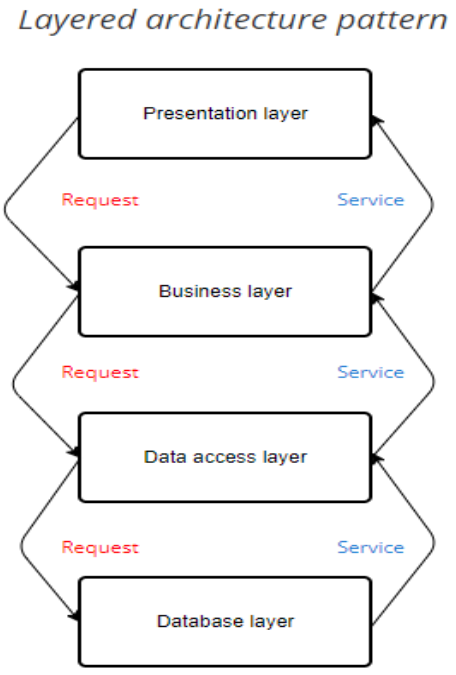


Image 4. Layered architecture pattern

3.2.3 MVC

MVC (Model-View-Controller) is a software architectural pattern (although it could be also considered a design pattern depending on the point of view) commonly used to implement user interfaces, data management and business logic. It advocates mainly for separation of concerns (that we described before), providing a better division of the core web app functionalities, making it easier to maintain and expand, improving testability and reusability.

There MVC is conformed by 3 parts:

1. Model:
    a. Represents the data and business logic of the application.
    b. Encapsulates data and the rules that govern how that data can be manipulated.
    c. May contain database access logic or interact with external APIs to retrieve and store data.
    d. Responds to requests from the controller to update the data.
2. View:
    a. Responsible for presenting the data to the user.
    b. Typically consists of user interface elements like HTML templates, CSS styles, and JavaScript code.
    c. Does not contain any business logic and shouldn't modify the data directly.
    d. Receives data from the controller and formats it for display.
    e. May send user interactions (like button clicks or form submissions) to the controller.
3. Controller:
    a. Acts as an intermediary between the model and the view.
    b. Receives user requests.
    c. Determines which part of the model needs to be updated based on the request.
    d. Updates the model and retrieves any updated data.
    e. Instructs the view to update itself based on the changes in the model.

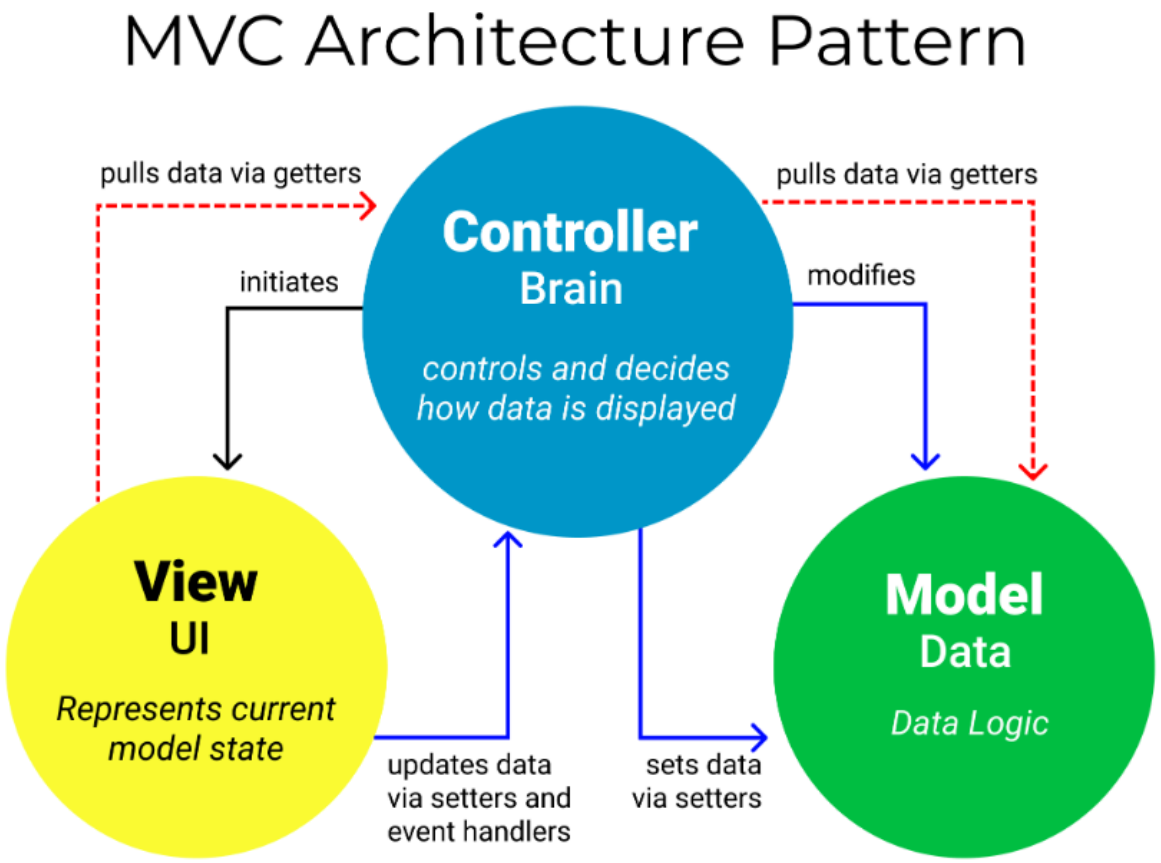


Image 5. MVC Architecture Pattern [5]

3.2.4 Monolithic architecture

For this project I am using a monolithic architecture. This means that the whole application is built as a single unit and that all of the code for the application is contained in one place.

This architectural decision was taken because this project would not be pretty big and it would be mainly handling pretty related services, making me decide to take a monolithic approach instead of, for example, a microservices approach, where each functionality of the project is independent and self-contained making it more suitable for larger project with different functionalities but increasing complexity and potentially losing efficiency.

3.3 Design patterns

A design pattern is a reusable solution to a common problem in software design. It helps solve the problems in a structured and standardized manner.

3.3.1 DAO

Data Access Object (DAO), is a structural design pattern that is used to decouple the application code from the database access. This pattern grants a layer of abstraction enhancing separation of concerns and loose coupling, making our app easier to maintain.

One of the core benefits of using this pattern is that, as we said previously, decouples the logic from the database access and management from the rest of our app logic, this makes that if for example in the future we want to make some changes in the database, all of our code is inside this DAO layer, avoiding us to have to check through the whole app to make the changes.

3.3.2 Singleton pattern

The Singleton pattern is a creational design pattern that ensures that a class has only one instance and provides a global access point to it. This is pretty helpful when we want, for example, to connect to a database, because we only want to have one active connection when working with the database. Without using this pattern it could be possible that multiple connections are established in one request flow generating a lot of potential problems.

3.3.3 Dependency Injection

Dependency Injection is a software development technique (not directly a design pattern although used in many of them) where an object receives the objects or functions that it needs in order to function, without having to take care about how they intrinsically work. This, also ensures loose coupling and improves separation of concerns improving the maintainability and testability of our app.

Here is a example of using Dependency Injection in the application:

```php
/**
 * CartCookie constructor.
 *
 * @param CartDataInitializer|null $cart The cart data initializer.
 * if null, the cart will be fetched from the cookie.
 * if not null, the cart will be initialized from the cart data initializer.
 */
4 references | 0 overrides
public function __construct(?CartDataInitializer $cart = null)
{
    if ($cart !== null) {
        $cartData = $cart->getProducts();

        foreach ($cartData as $product) {
            $productData = new CartCookieItem($product->getId(), $product->getQuantity
            ());
            $this->cart[] = $productData;
        }
    } else {
        $this->cart = self::getCartFromCookie();
    }
}
```

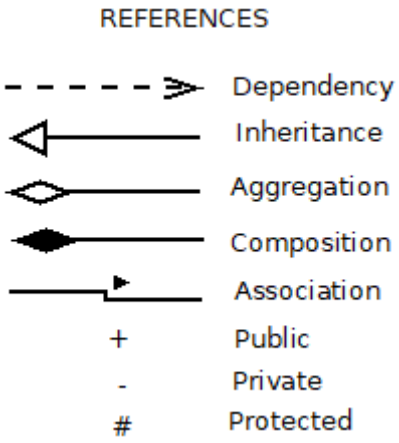
Image 6. Example of Dependency Injection in the application

In this code snippet we have the constructor method of the CartCookie class, it takes a CartDataInitializer instance as a parameter. If the cart is set, thus we receive an instance of CartDataInitializer, we execute the getProducts method in order to get the data from the object and then we construct the CartCookie cart attribute with this data. If there are no arguments given, the constructor calls its own static method getCartFromCookie.

As we can clearly see in this example the constructor does not care about how the CartDataInitializer handles itself in order to get the data the constructor need, it just knows that an instance of this class will be given, then the getProducts methods will do its magic inside the CartDataInitializer class, and then the constructor can build up the CartCookie cart attribute with it, the exact implementation details of CartDataInitializer are out of its scope and it only focuses on building the CartCookie.

3.4 Classes diagram

In this section I will expose the classes diagrams used for this application by functionality. I will take this approach because I think it is a clearer and more straightforward way of displaying the diagrams than simply exposing all the classes' diagram maps at once. Repeated classes will be indicated with a stereotype that will point to the section where the class definition is located.



3.4.1 Shared classes

In this section I will expose the class diagrams that are shared amongst some of the other classes, so when they are used in other classes I do not have to repeat the whole class again.

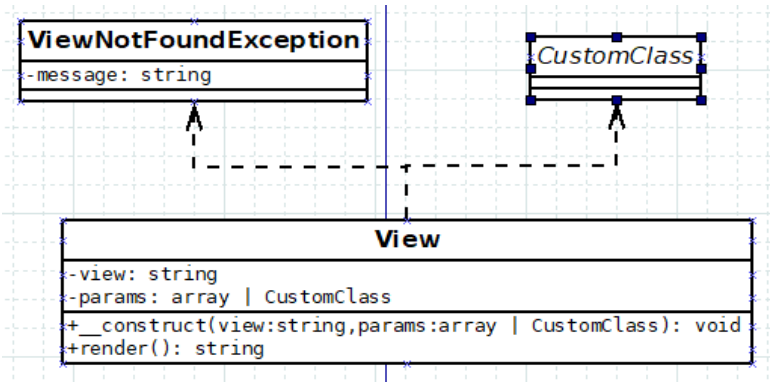


Diagram 5. View Class

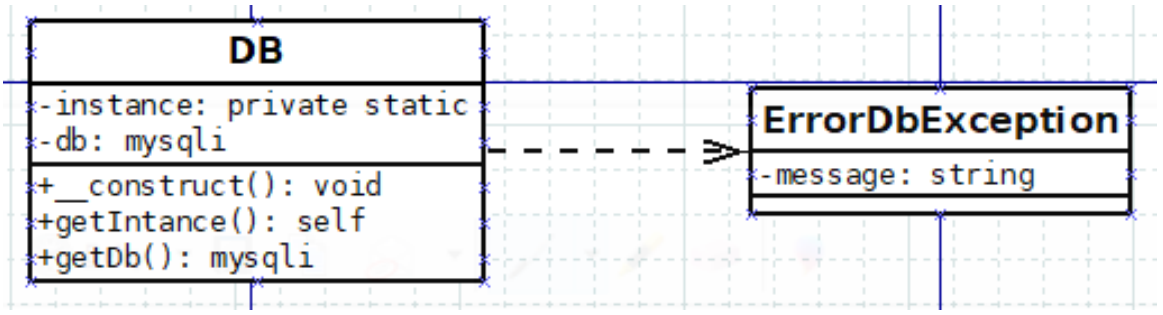


Diagram 6. DB class

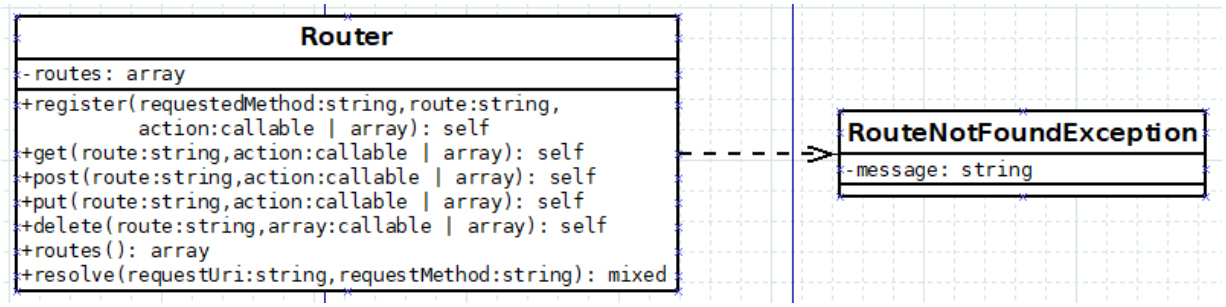


Diagram 7. Router class

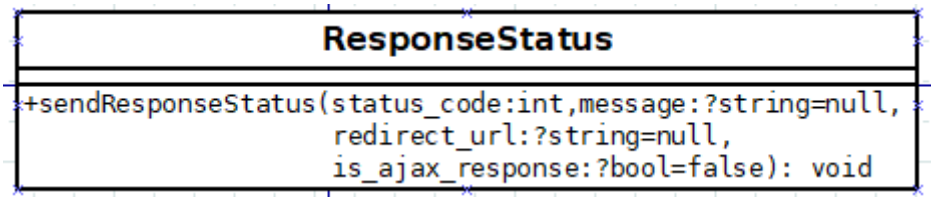


Diagram 8. ResponseStatus class

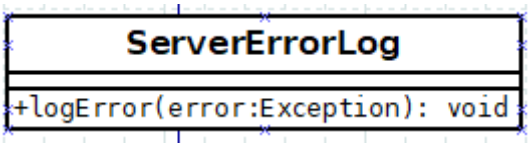


Diagram 9. ServerErrorLog class

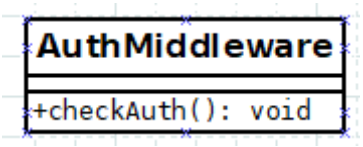


Diagram 10. AuthMiddelware class

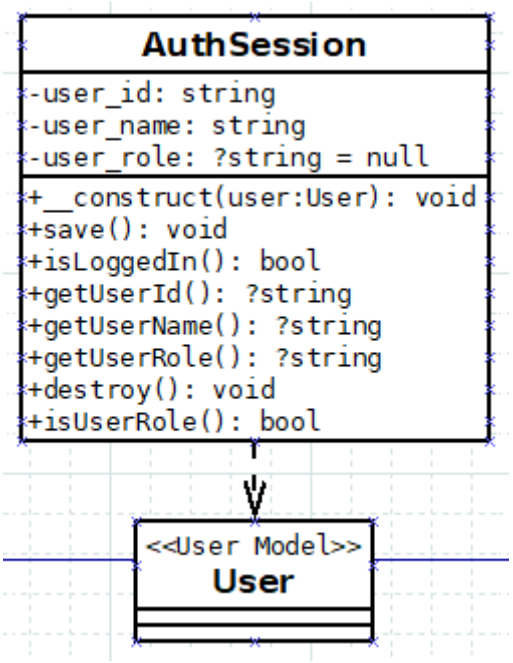


Diagram 11. AuthSession class

3.4.2 Models

In this section I will expose the class diagrams that belong to the model layer of the MVC pattern.

3.4.2.1 Product Model

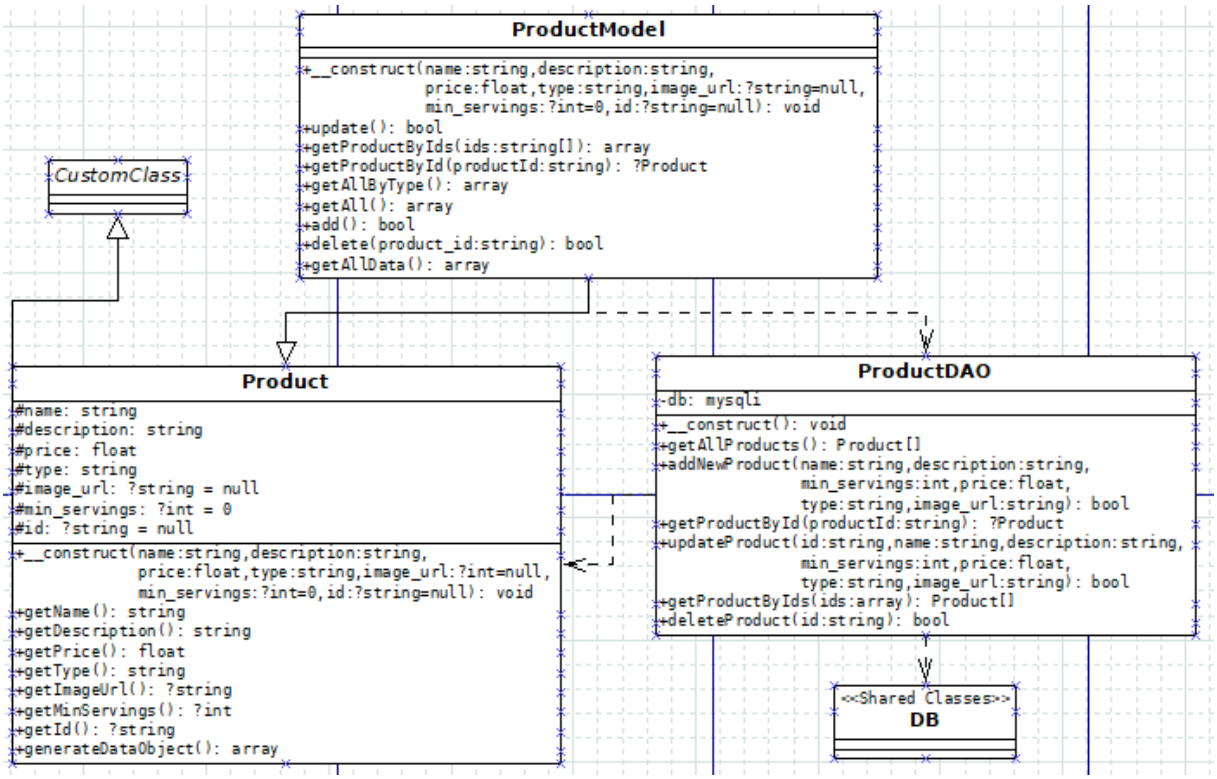


Diagram 12. Product Model

3.4.2.2 Cart Model

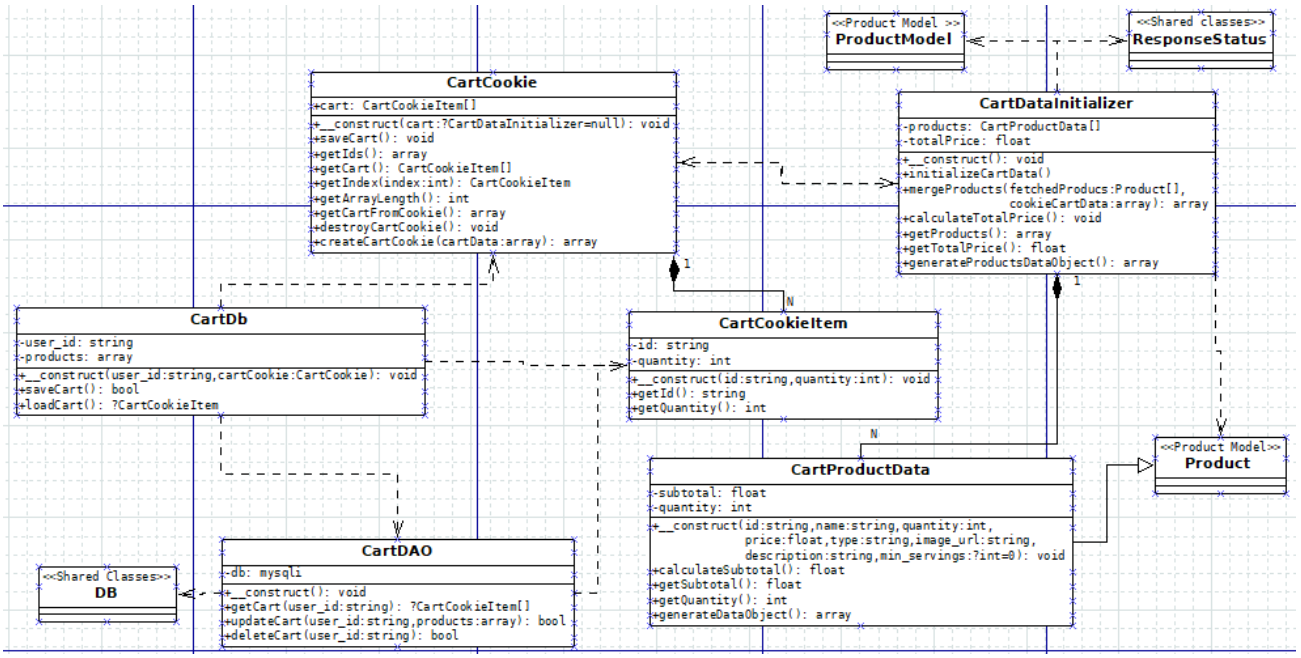


Diagram 13. Cart Model

3.4.2.3 User Model

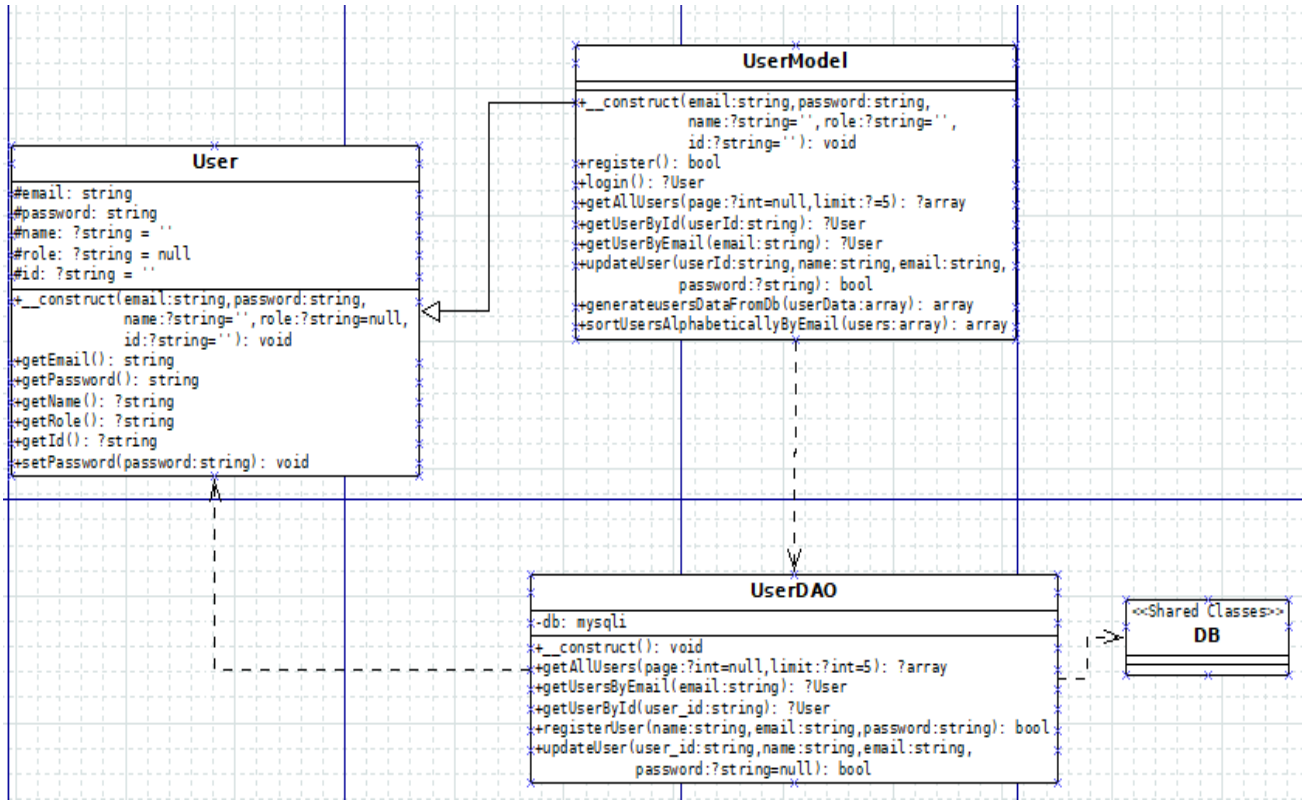


Diagram 14. User Model

3.4.2.4 Address Model

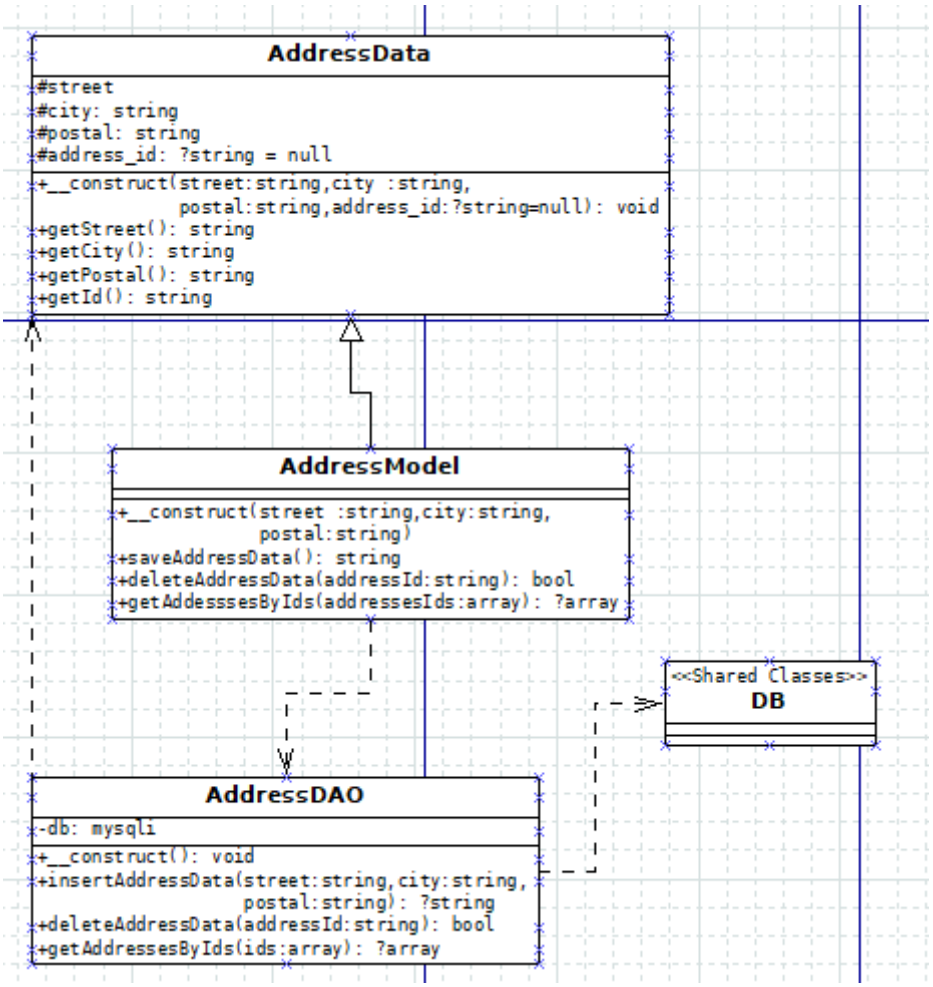


Diagram 15. Address Model

3.4.2.5 Order Model

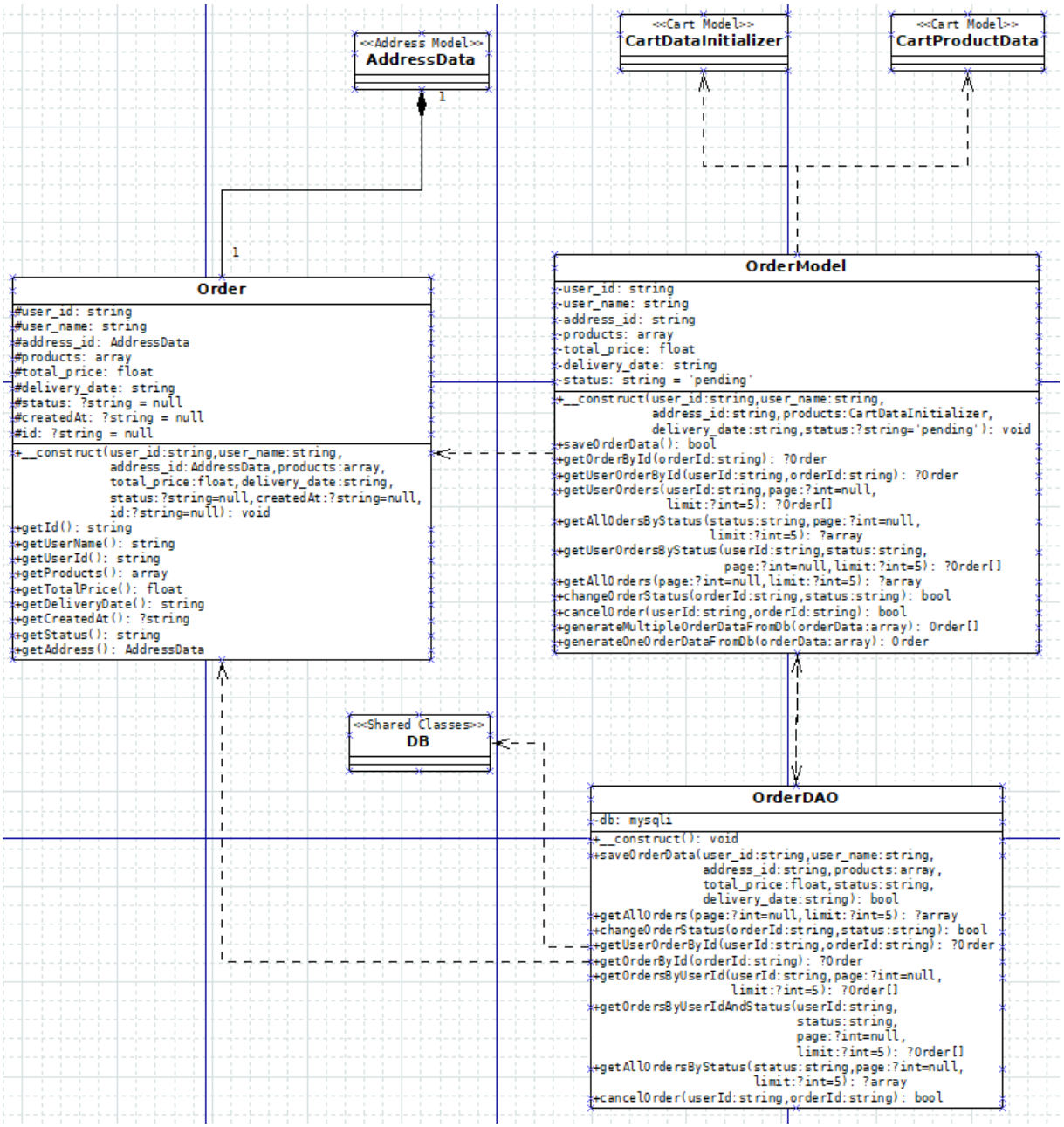
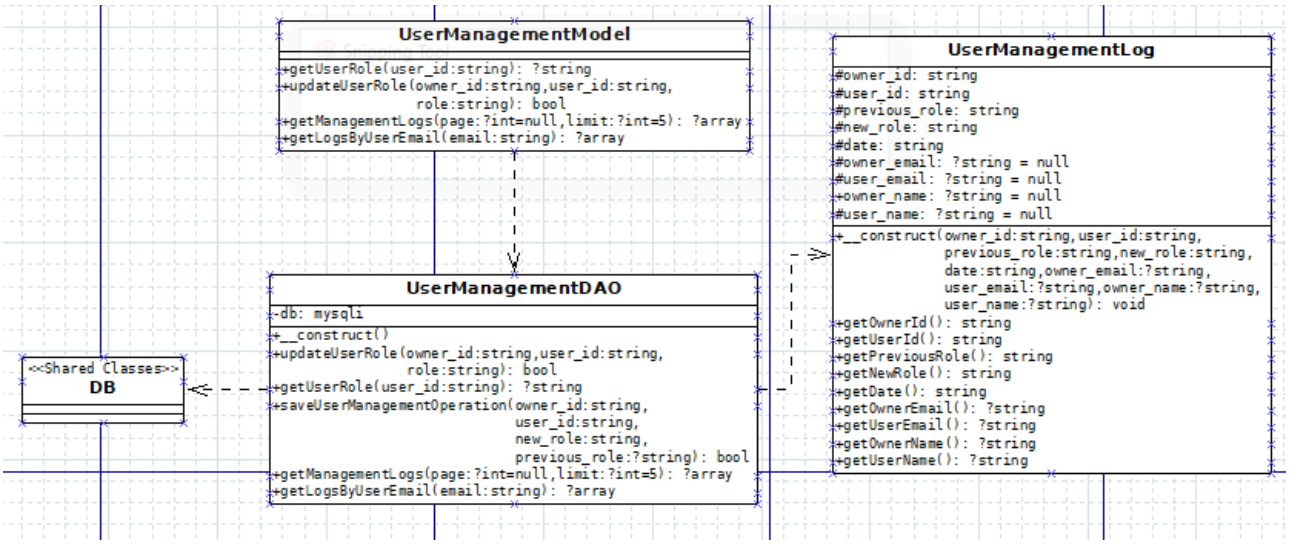


Diagram 16. Order Model

3.4.2.6 User Management Model



3.4.3 Controllers

3.4.3.1 Base Controller

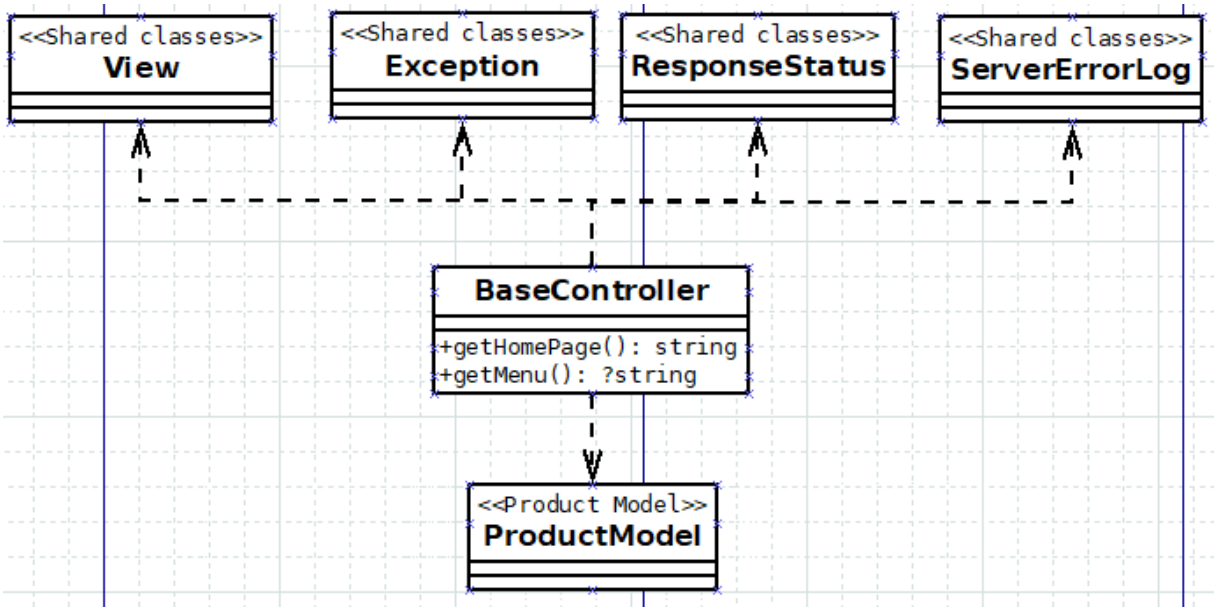


Diagram 17. Base Controller

3.4.3.2 Authentication Controller

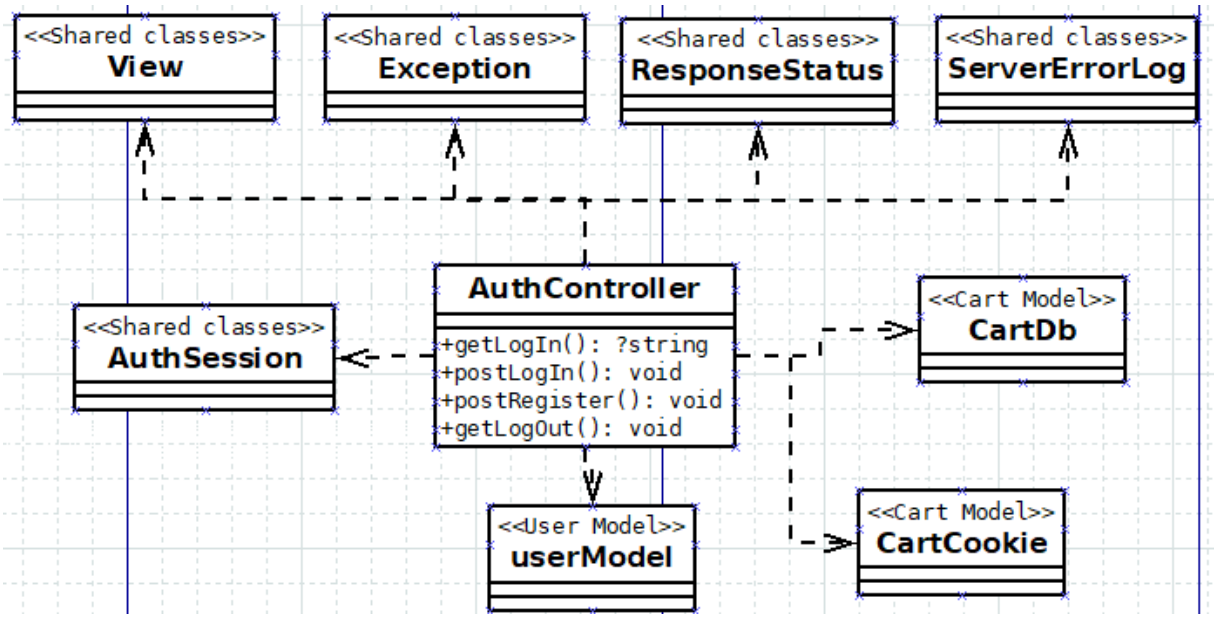


Diagram 18. Authentication Controller

3.4.3.3 Cart Controller

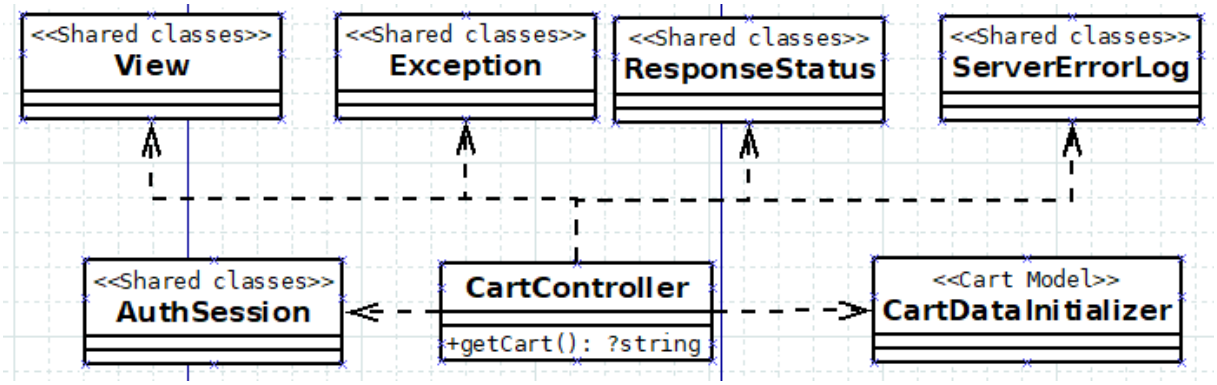


Diagram 19. Cart Controller

3.4.3.4 Order Controller

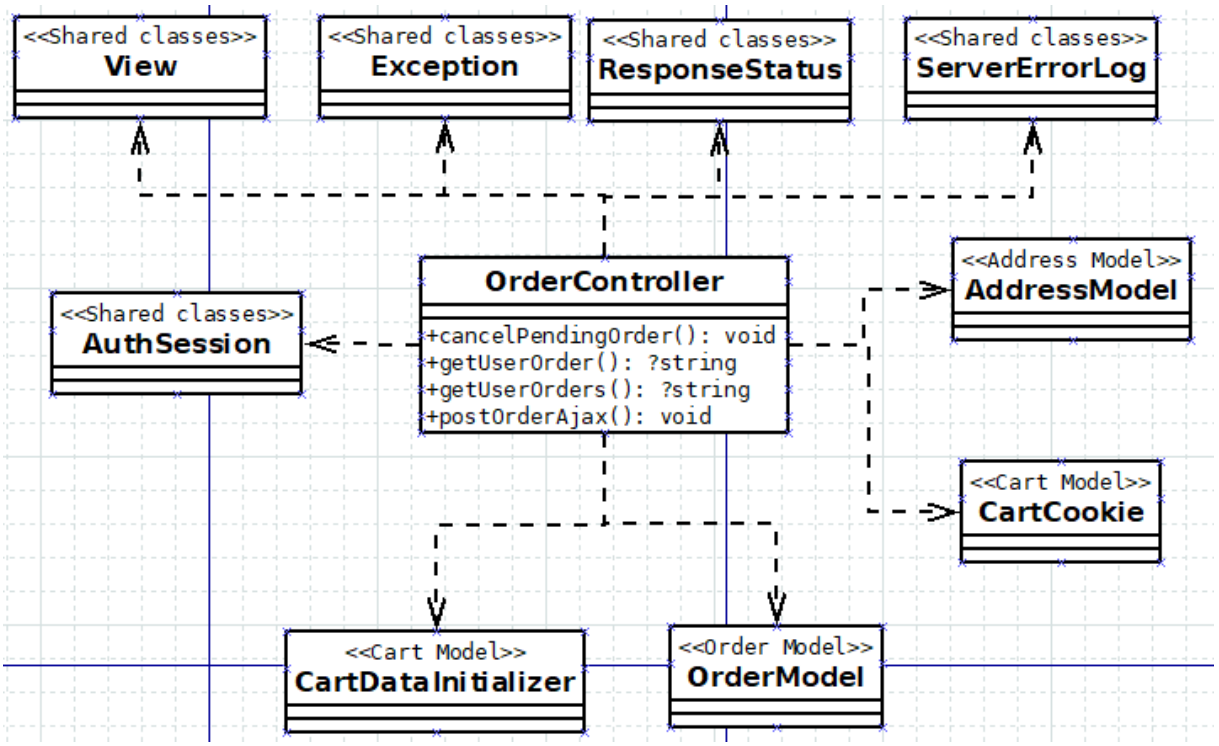


Diagram 20. Order Controller

3.4.3.5 User Controller

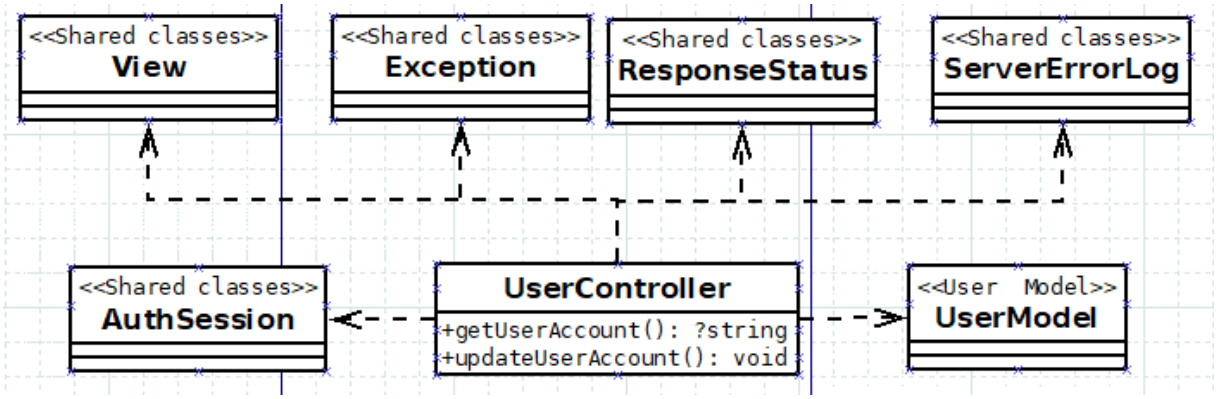


Diagram 21. User Controller

3.4.3.6 Admin Menu Controller

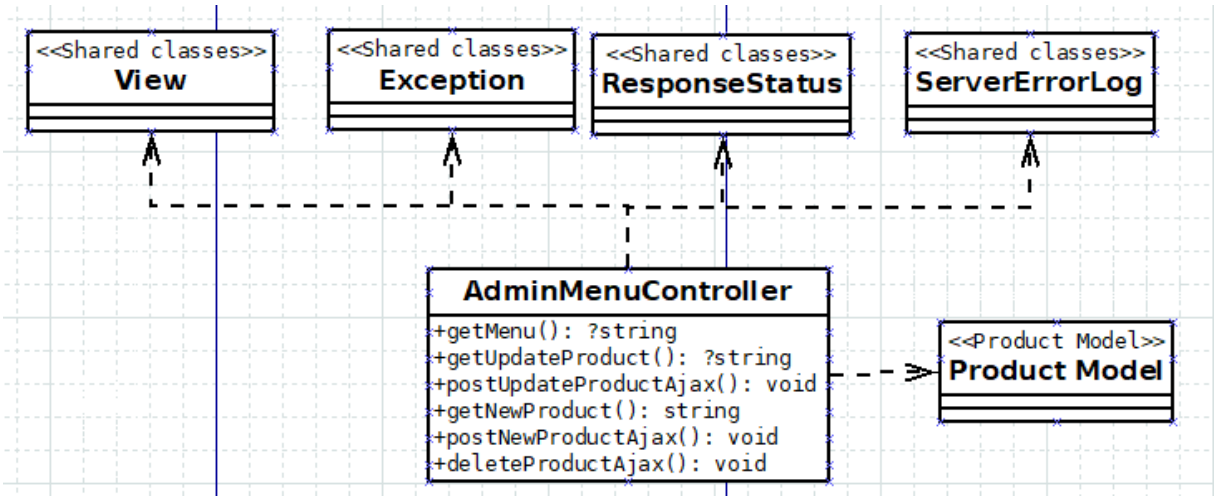


Diagram 22. Admin Menu Controller

3.4.3.7 Admin Order Controller

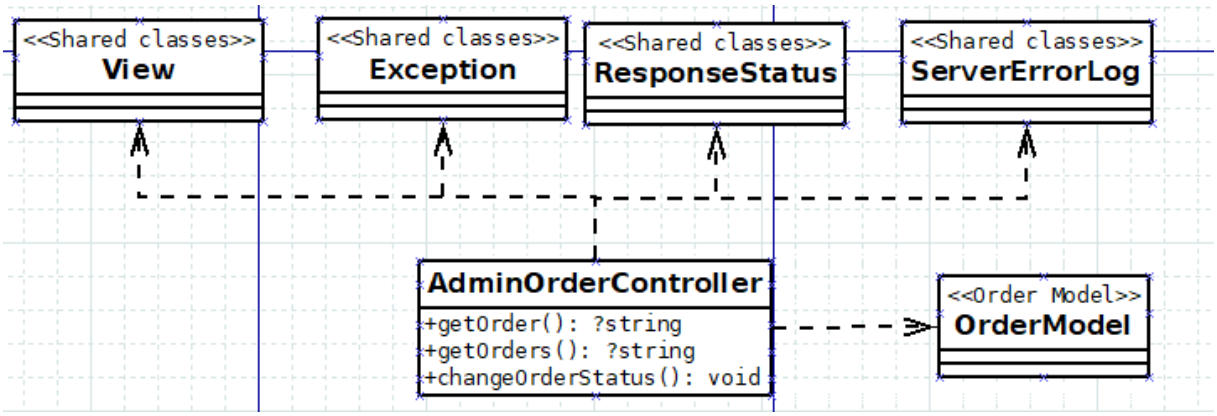


Diagram 23. Admin Order Controller

3.4.3.8 Owner User Controller

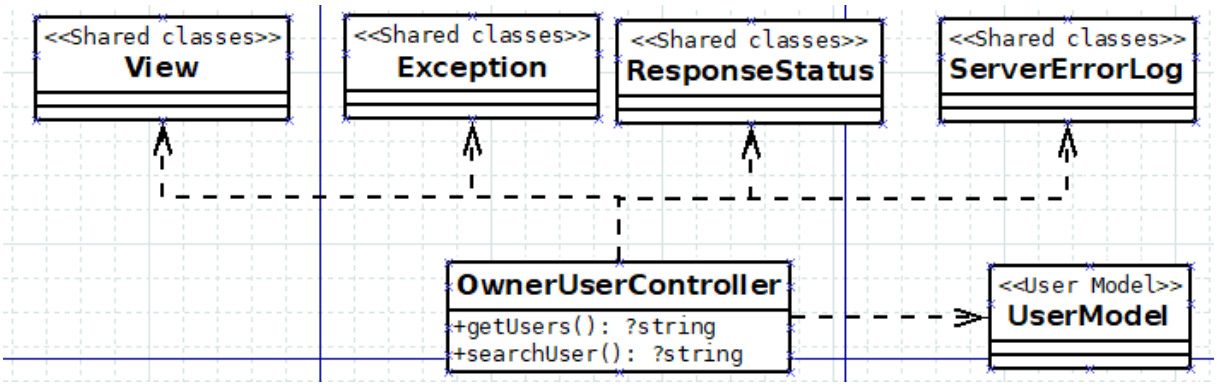


Diagram 24. Owner User Controller

3.4.3.9 Owner User Management Controller

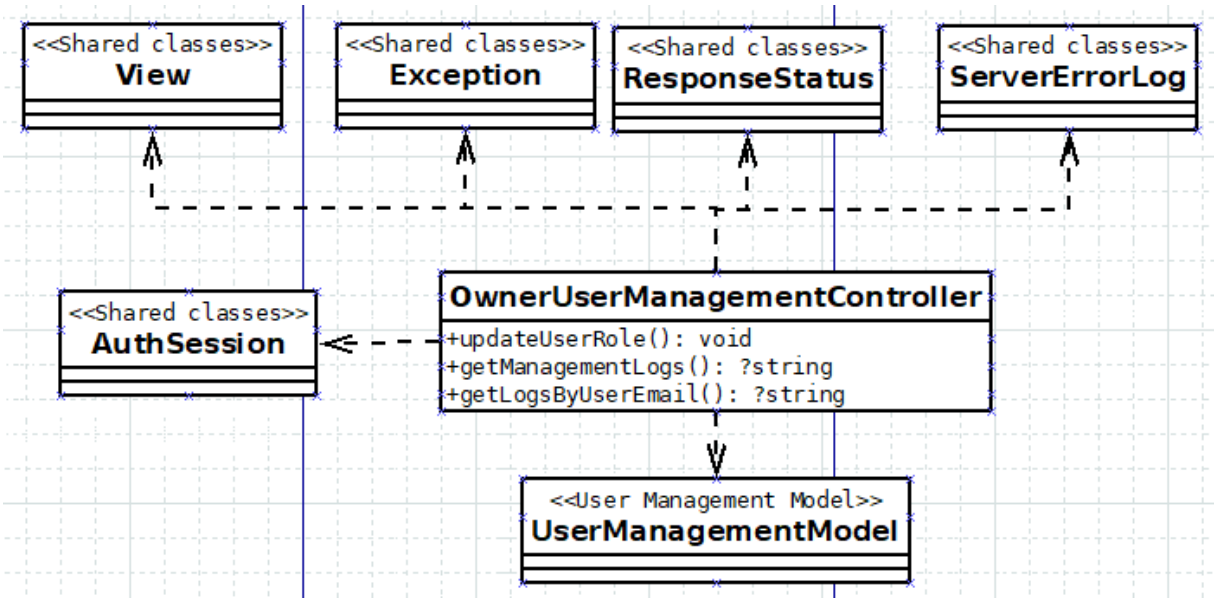


Diagram 25. Owner User Management Controller

3.5 Database design

In this section the database design will be explained but first we will point out what makes a database design good and which principles it should respect:

- Data integrity: the database design must ensure the consistency and integrity of the data.
- Data minimization: the database design must avoid storing unnecessary data.
- Normalization: normalization is the process of organizing the data tables in order to improve data integrity and reduce redundancy.
- Performance: the database design must use optimized queries in order to be as performant as possible. This can be improved, for example, with the indexing of frequently required data.
- Scalability and maintainability: the database should be easy to grow in size and complexity, so that when the application scales the database scales accordingly.
- Security: all sensitive information must be protected and should not be reachable from unauthorized users, passwords should always be encrypted and they should never reach the client.

For the database design we will use an Entity-Relationship Diagram (ER). An ER diagram is a type of flowchart used in database design. Its goal is to visually define the relationships (relationship) between the different tables (entities) of our database and to guarantee that the design principles should be respected.

This diagrams are created using 4 main concepts:

- Entity: entities represent the real entity (users, orders, carts) about which the information is going to be stored.
- Attributes: attributes are the specific properties that describe the entity, for an user it could be email, password, role, etc.
- Relationship: relationships define the connections/relationships between the entities, they show how entities are correlated to each other and they are represented by a verb.
- Cardinality: refers to the number of occurrences that one entity can be associated with another entity given a relationship.

For this project I started with a more theoretical ER diagram, but as my application grew I realized that I could come up with a better design, even if it is not such a standard and theoretical one in terms of design. This theoretical vs custom design will be explained in the following two sections.

3.5.1 Theoretical design

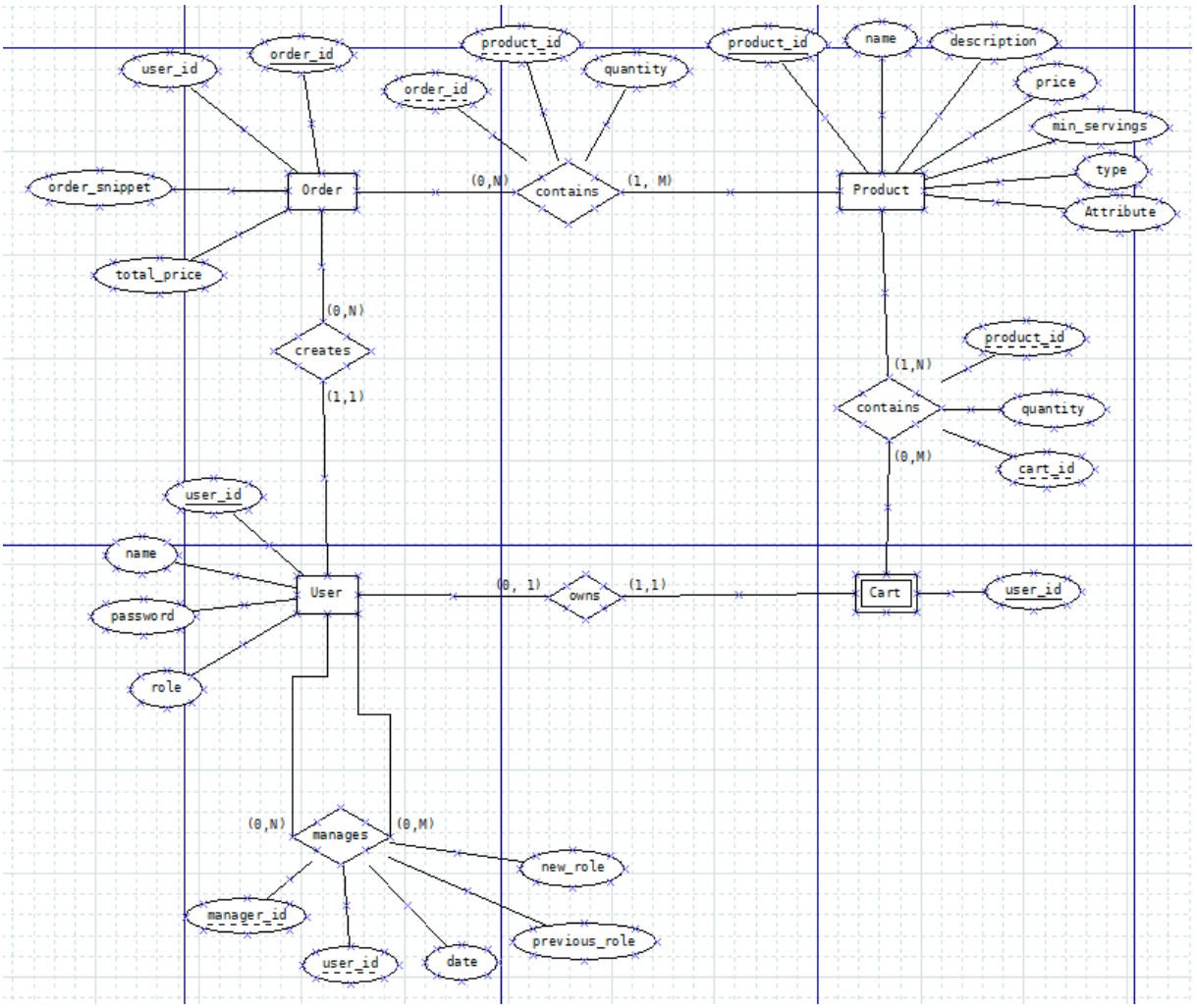


Diagram 26. Prototype ER diagram

3.5.2 Final design

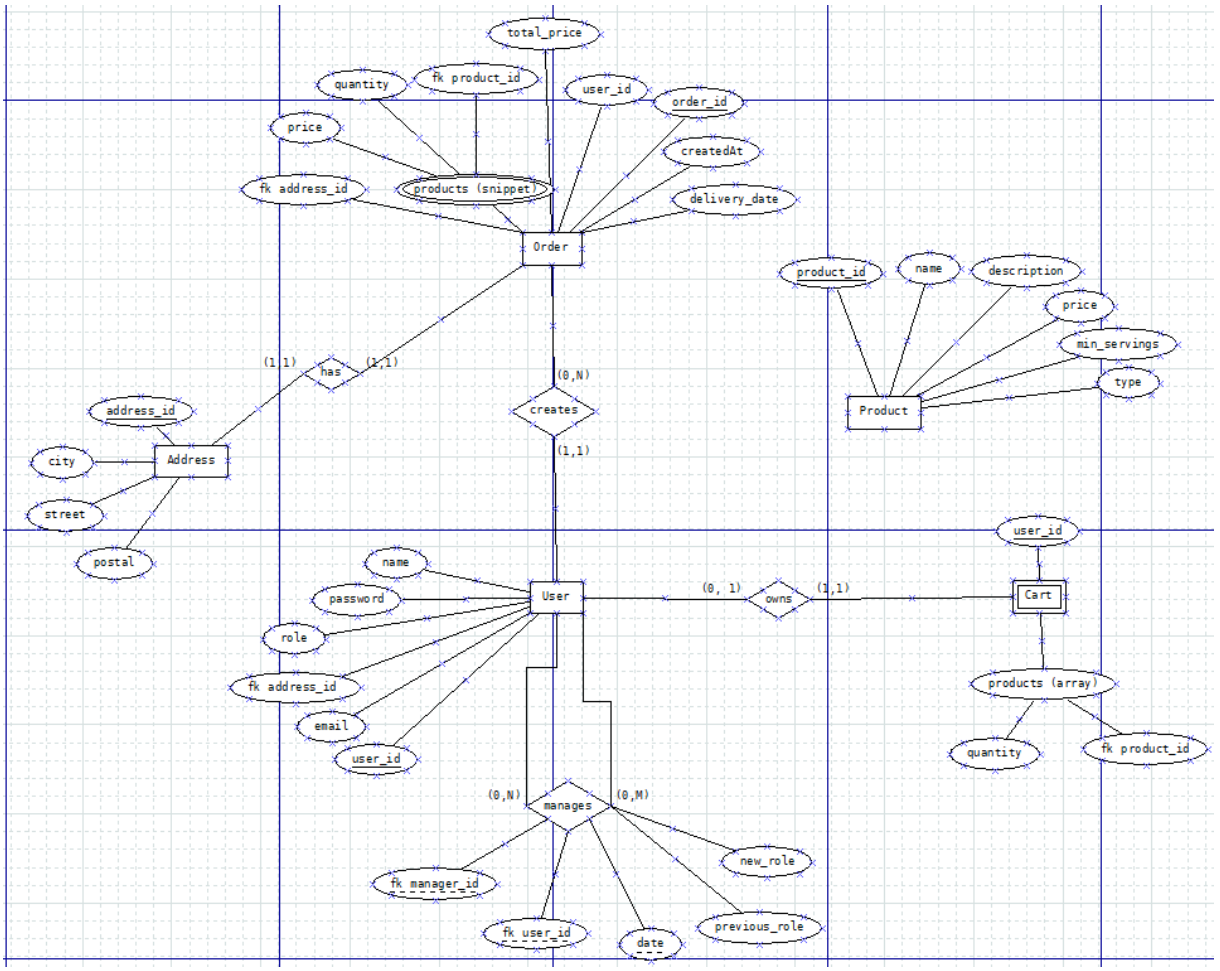


Diagram 27. Final ER diagram

3.5.3 Theoretical prototype vs final design

The difference between both designs rely on the Cart-Product relationship and the Order-Product relationship.

3.5.3.1 The Order-Product relationship

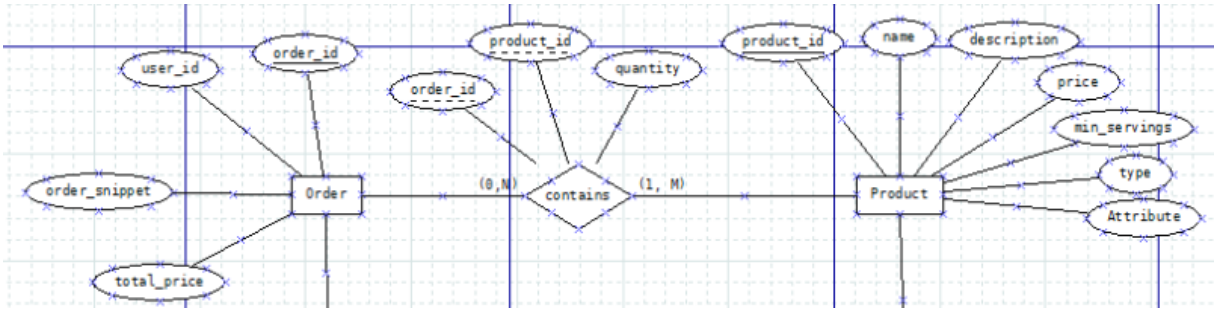


Diagram 28. Prototype Order-Product relationship

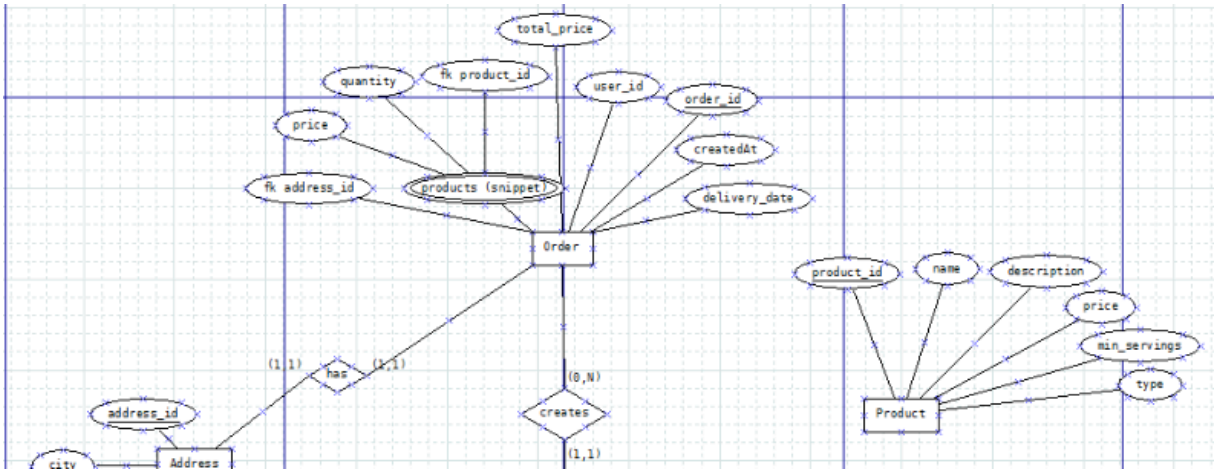


Diagram 29. Final Order-Product relationship

As we can see the main difference is that there is no actual relationship in the diagram between the Product entity and the Order entity, and that makes total sense when we understand what an order actually is.

An Order is nothing more than a snippet of a set of products data and other attributes in the moment when the user placed the order, so it actually is not directly related to the Product entity because, once

the order is placed, it is no longer related to the products in the database as it is a snippet from the past state.

Why is that? If we would keep the Order-Product relationship, every change in the products after the order has been placed would affect all of our orders in the database, because they are related to them. This will make sure that the orders will actually be modified after they were placed, referencing completely wrong data because the order data when placed has been modified by an external source out of any wanted input.

To avoid this a more collection based NoSQL approach has been taken inside of our Order entity, where the data from the order is actually checked and purged before the order has been placed via the CartDataInitializer class, which makes sure that the data integrity is being preserved, and once that checks have been done and the order data has been handled accordingly, we save a snippet of this order in the database with the needed data. This would make an order a self-contained collection of information that is protected against external changes in the rest of the database. The product ids are saved in the case that the user wants to check the specific product page (which is not implemented yet), so that we can retrieve the necessary information of the product and show it again to the user.

3.5.3.2 The Cart-Product relationship

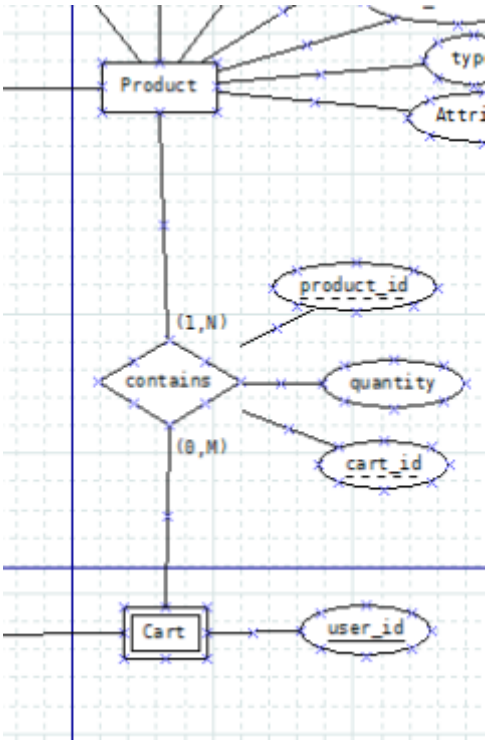


Diagram 30. Prototype Cart-Product relationship

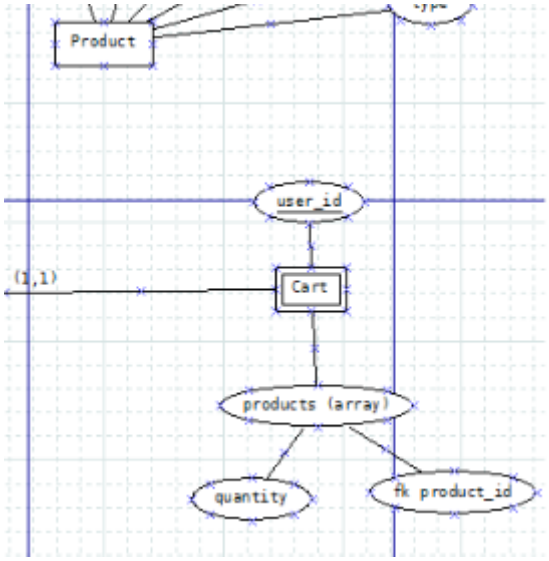


Diagram 31. Final Cart-Product relationship

In this case the main difference is also that there is no direct relationship in the diagram between the Cart entity and the Order entity. This approach has been taken to improve the performance of the application because as in the Order-Product relationship, a N:M cardinality will require a relationship table. The Cart entity function is to store the data from the cart cookie when the user logs out, so that his cart cookie data can be stored and persisted between sessions.

If we take the theoretical approach, everytime we make a change in the cart it would have to be reflected in the Cart-Order relationship table.

Let's say for example that each customer cart has 5 items and we have 100 customers, this will sum up to a total of 500 rows in the relationship table, that means that in every request the database would have to search between 500 rows and check them against the data from the Products table in order to preserve data integrity. Given that for example, the user inserts a new product in the cart, updates another one and deletes another, it would mean that these 3 different queries will have to perform in a 500 rows scale for the given case. This could easily lead to performance issues once the application scales as well as increase the number of queries and its complexity.

For solving this problem also a collection based NoSQL approach has been taken. As with the Order-Product relationship, the CartDataInitializer is the one that validates the data against the products table when the cart data from the database is loaded.

In this case we have a single row for each customer, in the example given above would mean that we have 100 rows. When the user logs out, the cart cookie data is stored as a ‘collection’ in the Cart entity without being validated (validating here makes no sense as the data from the products can change while the user is logged out and everytime the user opens the cart page the data is already being purged and validated) meaning we only have to perform one `INSERT ... ON DUPLICATE KEY UPDATE` to update or insert the cart data or one delete operation when the cart is empty in a 100 rows database. When the user logs in again we validate the data thanks to the CartDataIntializer class in order to maintain the integrity and we store the validated data in the cart cookie.

This completely boosts the speed at which the database functions and reduces the number of queries necessary compared to the theoretical approach and that is why this approach was taken. We still maintain the data integrity thanks to our CartDataInitializer class, we store less data and we boost the performance.

3.6 Request flow

In this section we will visually represent the flow followed by a request made to our server.

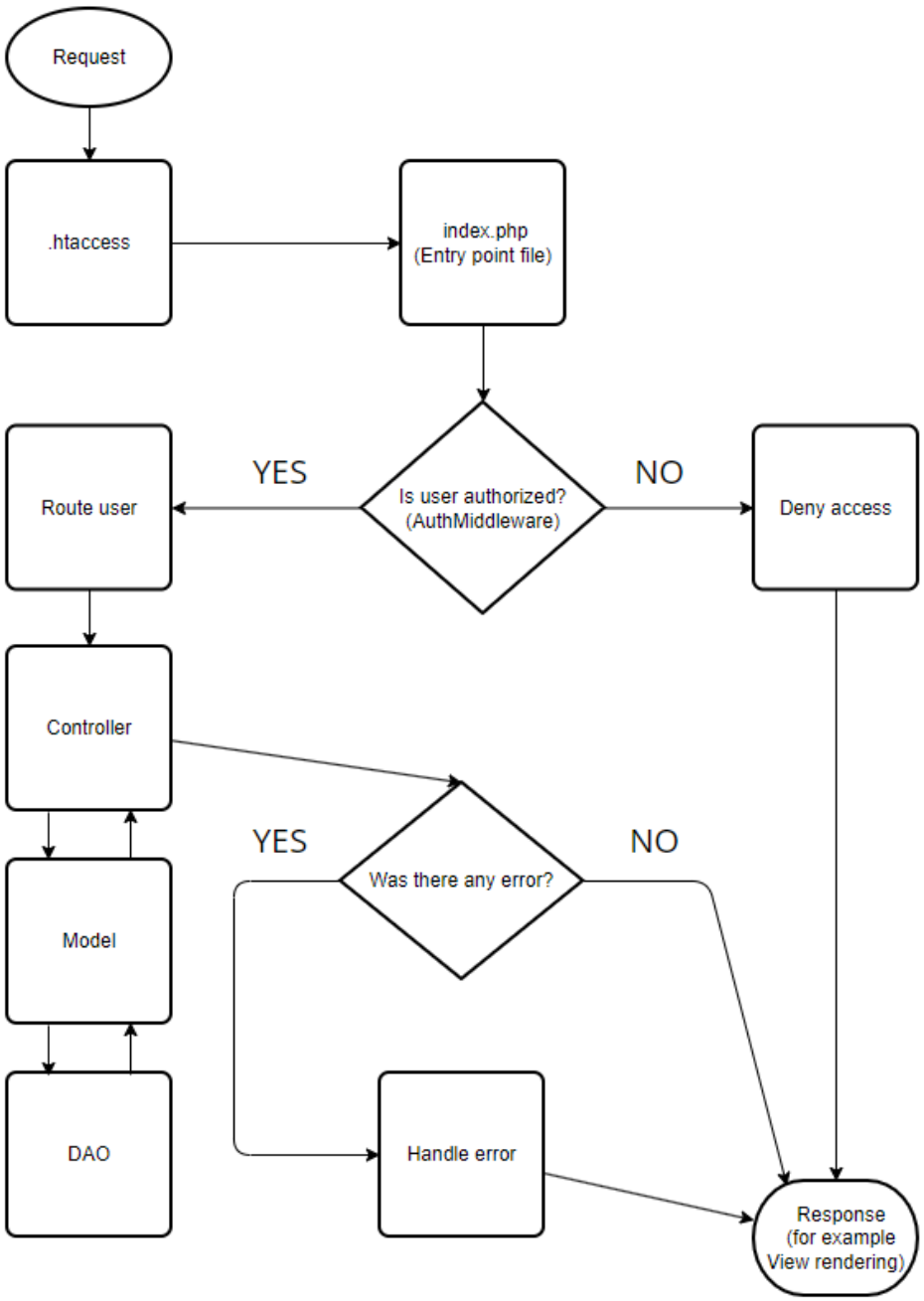


Diagram 32. Request flow chart

4. Implementation

In this section we will expose technologies and tools that have been used to create the project. All the information regarding deployment hardware and software will be explained in a different and independent Deployment section in page 91 .

4.1 Tech stack

4.1.1 Front-end

HTML5:

HTML5 is the latest version of Hyper Text Markup Language (HTML) and is one of the cores in web development. Usually referred to as the skeleton of a webpage, its function is to structure and organize the webpage in a meaningful and semantical manner with the usage of the specific markup tags. These semantic tags help the crawlers in the web index the web pages to later position them in the browser searches.

In comparison with its predecessor versions, HTML5 was designed to enhance storage, multimedia and hardware access in a web that with the years have increased its interactivity needs[6].

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
</head>
<body>

</body>
</html>
```

Image 7. HTML code snippet

CSS3:

CSS3 is the latest Cascading Style Sheets (CSS) standart. It is the language used to define the visual presentation and layout of webpages. It works alongside HTML, which structures the content, to create the final webpage that users see.

For this particular webpage I wanted to experiment a bit, so I am widely using CSS native nested selectors in order to follow a more functional component-like approach.

```css
#side-menu {
  z-index: 99;
  position: fixed;
  top: 2rem;
  right: 0;
  background-color: var(--background-primary);
  padding: 3rem 2rem;
  font-size: 1.5rem;
  & h4 {
    font-size: 2rem;
  }
  & ul li a {
    padding: 0.3rem 0;
    display: flex;
    gap: 1rem;
    align-items: center;
  }
}
```

Image 8. CSS3 code snippet

JavaScript:

JavaScript is the interpreted scripting programming language that is used in front-end development to add interactivity to the webpage as well as to make AJAX requests (explained in page 53). Nowadays thanks to the V8 Google engine it can also be used in order to program backend applications.

This interactivity is obtained by executing JavaScript scripts in our HTML that add event listeners to the webpage DOM (Document Object Model) which is a representation of the web page structure.

Once these event listeners are triggered we can then manipulate the DOM to add interactivity or we can also make AJAX requests to make a fetch request without having to reload the browser.

Image 9. JavaScript code snippet

4.1.2 Back-end

PHP:

PHP (Hypertext Preprocessor) is an open source interpreted server side scripting language that runs in the server and is widely used for back-end web development. PHP is used to leverage our application server in order to generate dynamic content, handle user interactions, connect to databases, manage data, all in the server side. It is also pretty comfortable to use because it allows us to write scripts directly into the HTML, making our front end more tailored and dynamic (although we have to be careful and respect separation of concerns, meaning that this scripts should primarily be used in order to display or organize certain information and not execute business logic, which is part of other layer of the application).

src > front > views > cart.php > ...

```php
<?php require_once __DIR__ . '/includes/shared-head.php'; ?>

<link rel="stylesheet" href="/src/front/css/cart.css">
<script type="module" src="/src/front/scripts/cart/add-one.js" defer></script>
<script type="module" src="/src/front/scripts/cart/remove-from-cart.js" defer></script>
<script type="module" src="/src/front/scripts/cart/remove-one.js" defer></script>
<script type="module" src="/src/front/scripts/cart/create-order.js" defer></script>

<?php echo generateSEOTags('Cart', 'This is the cart page, here you can find all the products that you have added to your cart.'); ?>
</head>

<body>
    <?php require_once __DIR__ . '/includes/shared-components.php'; ?>
    <?php
    echo createHeader();
    ?>
    <main>
        <h1 id="page-title">Cart</h1>
        <div class="body-container">
            <?php include_once __DIR__ . '/includes/cart/left-container.php' ?>
            <?php include_once __DIR__ . '/includes/cart/right-container.php' ?>
        </div>
    </main>
    <?php require_once __DIR__ . '/includes/footer.php';
```

Image 10. PHP code snippet

MySQL:

MySQL is an open-source relational database management system (RDBMS) widely used in back-end development. It provides a structured way to store, organize, and retrieve data for web applications using a SQL database architecture.

The database is accessed via queries to perform CRUD (create, read, update, delete) operations. MySQL integrates with various back-end programming languages for data-driven web applications. This can be seen in PHP where you can use the *mysqli* class in order to perform any wanted operations, from connection and disconnection of the database, as well as executing statements.

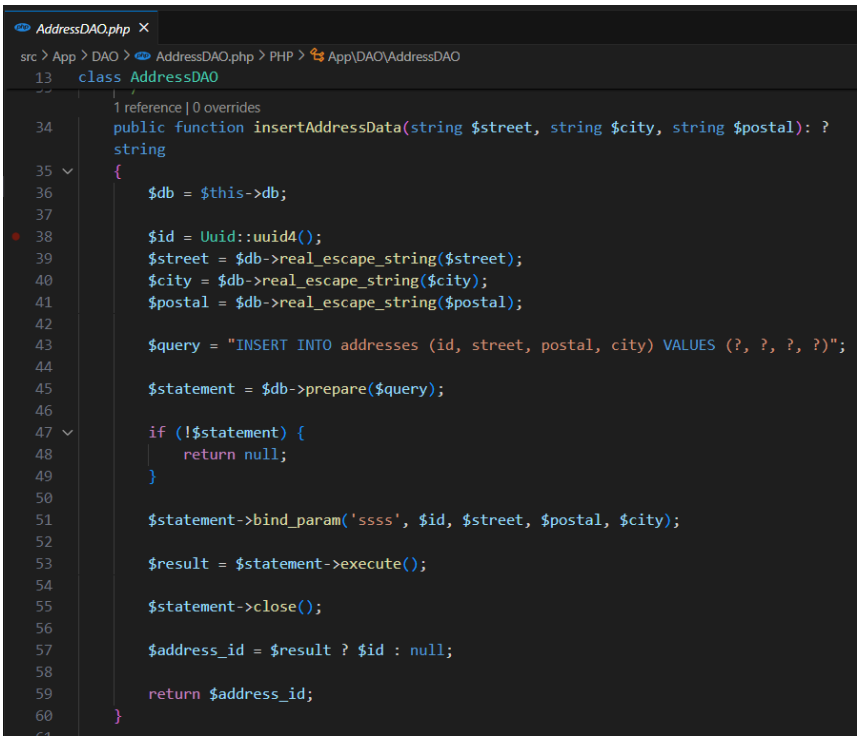


Image 11. Usage of the mysqli class to execute statements ($db is an instance of mysqli connection)

4.1.3 Data exchange

XML:

Extensible Markup Language (XML) is a markup language designed for storing and transporting data in a structured and human-readable format. It provides a versatile way to represent and exchange data between different applications and systems. We have mainly used XML in our SVGs files because for data transfer we have opted for the more modern JSON format that we will talk about next.

JSON:

JSON (JavaScript Object Notation) is a modern data format used for information exchange. Compared to XML is a more concise and lightweight approach. JSON data is organized into key-value pairs, nested objects and arrays, allowing it to represent complex data.

composer.json > ...

Install | Update | Check

```
{
  "require-dev": {
    "phpunit/phpunit": "^11" (11.0.9)
  },
  "autoload": {
    "psr-4": {
      "App\\": "src/App/"
    }
  },
  "autoload-dev": {
    "psr-4": {
      "Tests\\": "tests/"
    }
  },
  "require": {
    "vlucas/phpdotenv": "^5.6" (v5.6.0),
    "ramsey/uuid": "^4.7" (4.7.5)
  }
}
```

Image 12. JSON

4.1.4 Development techniques

AJAX:

AJAX (Asynchronous JavaScript and XML) is a set of development techniques that enables web applications to be updated asynchronously by communicating with the server without reloading the entire page. This allows for a more fluid and responsive user experience.

With the execution of a JavaScript script the frontend makes a fetch request to the server, which brings back the information required, and then this same script handles the information and updates the frontend acordingly. Traditionally this communication has been done using XML but with the years and the introduction of JSON it has replaced XML as the go data format for the information transfer between the backend and the frontend.

src > front > scripts > admin > delete-product.js > ...

```
const deleteButtons = document.querySelectorAll('.delete-button');

for (const deleteButton of deleteButtons) {
  deleteButton.addEventListener('click', async (e) => {
    e.preventDefault();

    const productId = e.target.dataset.productid;

    //encoding the image url to be able to send it as a query parameter
    const encodedImageUrl = encodeURIComponent(e.target.dataset.imageurl);

    let response;
    try {
      response = await fetch(
        `/admin/product/delete?productId=${productId}&imageUrl=${encodedImageUrl}`,
        {
          method: 'DELETE',
        }
      );
    } catch (error) {
      alert('Something went wrong, please try again later');
      return;
    }

    const responseData = await response.json();

    if (!response.ok) {
      alert(responseData || 'Something went wrong, please try again later');
      return;
    }

    const listElement = deleteButton.closest('.menu-list-item');

    listElement.style.display = 'none';
  });
}
```

Image 13. AJAX request

This script handles the deletion of menu items from the menu page, it adds event listeners to all elements that have the class delete-button. When the delete button is clicked we parse the needed data from the triggered button and we make the asynchronous fetch request to the backend. The backend processes the request and sends back a response and if everything went as expected the list item that contains the product is hidden, if not an error message will be displayed to the user. This has allowed the user to delete a product from the database, display possible errores and hide the element from the frontend without even reloading the page.

Funtional components:

A functional component is basically a function that returns HTML or in some frameworks like React, JSX. They are pretty helpful for building blocks of our code that can be reusable and composable, reducing the boilerplate, drastically increasing the easiness and speed of development as well as providing a layer of abstraction and improving separation of concerns, as each component can focus on its specific responsibility.

In the app there are two pretty basic vanilla PHP functional components that I built in order to avoid code repetition and leverage the advantages that I exposed before.

src > front > components > headerComponent.php > ...

```php
<?php

declare(strict_types=1);
17 references
function createHeader(?string $midHeader = null, ?string $rightHeader = null)
{
    if (!isset($rightHeader)) {
        if (isset($_SESSION['user']) && isset($_SESSION['user']['role']) && ($_SESSION['user']['role'] === 'admin' || $_SESSION['user']['role'] === 'owner')) {
            $rightHeader = '<li><a href="/admin/orders"> <img class="icon" src="/src/front/assets/icons/read-book-icon.svg" alt="menu-icon"></a></li>
                <li><a href="/admin/menu"> <img class="icon" src="/src/front/assets/icons/bag-icon.svg" alt="menu-icon"></a></li>';
        } else {
            $rightHeader = '<li><a href="/menu"> <img class="icon" src="/src/front/assets/icons/read-book-icon.svg" alt="menu-icon"></a></li>
                <li><a href="/cart"> <img class="icon" src="/src/front/assets/icons/bag-icon.svg" alt="menu-icon"></a></li>';
        }
    }
    return <<<HTML
     <header id="header">
        <div>
            <h1><a href="/">IB</a></h1>
        </div>
        <nav>
            <ul id="mid-header">
             $midHeader
            </ul>
        </nav>
        <nav>
            <ul id="right-header">
              $rightHeader
                <li id="hamburger-button"><span></span><span></span><span></span></li>
            </ul>
        </nav>
    </header>
HTML;
}
```

Image 14. Functional component

This is the first component and has made it possible to use this only single piece of code to write all the headers of our app with one single component. It takes two arguments, the midHeader and rightHeader, if the arguments are given the header will create the header with the customized mid and right header elements, but if for some reason no arguments are given there are some default components that will be displayed instead. As said before, that has allowed that only with the execution of this function I could create all the header of the webpage with tailor made details.

The header can be then called like this:

```php
<body>
    <?php require_once __DIR__ . '/../../includes/shared-components.php'; ?>
    <?php
    $midHeader = '<li class="btn-primary"><a href="/admin/product/new">New Dish</a></li>';
    echo createHeader($midHeader);
    ?>
    <main>
```

Image 15. Execution of the functional component

The other functional component created has been created for creating the SEO friendly head tags.

src > front > components > generate-SEO-tags.php > ...

```php
<?php
declare(strict_types=1);
17 references
function generateSEOTags(?string $title = null, ?string $description = null)
{
    if (!isset($title)) {
        $title = 'Chef Manuelle Webpage';
    }

    if (!isset($description)) {
        $description = 'Chef Manuelle is a culinary virtuoso with over 25 years of
        culinary expertise. Her journey in the world of gastronomy has been nothing short
        of extraordinary. Having honed her skills in the most prestigious kitchens around
        the globe, she now stands at the threshold of a new culinary adventure.';
    }


    $metaDescription = '<meta name="description" content="' . $description . '">';
    return <<<HTML
     <title>$title</title>
    $metaDescription
HTML;
}

```

Image 16. Another functional component

The execution of this header creates the customized title and description and can be executed in the code like this:

src > front > views > menu.php > ...

```php
<?php require_once __DIR__ . '/includes/shared-head.php'; ?>

<link rel="stylesheet" href="/src/front/css/menu.css">
<script type="module" src="/src/front/scripts/menu/add-to-cart.js" defer></script>
<script type="module" src="/src/front/scripts/menu/button-scroll-menu-page.js" defer></script>
<script type="module" src="/src/front/scripts/menu/x-draggable-container.js" defer></script>

<?php echo generateSEOTags('The Menu', 'This is the menu page, here you can find all the delicious dishes that our chef prepares for you.'); ?>
</head>
```

Image 17. Execution of the functional component

It could be further improved for adding more SEO metadata in the functional component but as the goal of this project is to create a prototype and not a production ready 100% SEO performant web page I have only included the basic ones to showcase how it works.

4.2 Tools used

Visual Studio Code (VS Code): it is the code editor that has been used to code this project.

Composer: this is the package manager that has been used for this project. I have used it for using its Autolader features as well as for the Ramsey UUID package for generating random ids and VLucas dotenv to access environmental variables.

Git: Git is the version control software used for this project. With it I could control and manage the repository code.

Github: Github is the online platform used for online version control.

Dia: Dia is the software that used for some of the UML diagrams, the rest of them have been created with Online Visual Paradigm at https://online.visual-paradigm.com

MySQL Workbench: this is the software used for creating and managing our MySQL databases.

Online Gantt: for the gantt diagram: https://www.onlinegantt.com/

4.3 Implementation details

In this section we will dive into how some of the features were implemented in the project. I will not explain all of the features that the app provides because this will be too long and it is not the main focus of this document. If you want to know anything how the app was implemented and check the code you can visit the following Github repo: https://github.com/josemontano1996/nebrija-proyecto-integrador-1

I will showcase how a logged in user can add products from the menu and place an order. I have chosen this one because it is one of the more complex features implemented, it uses AJAX requests, cookie storage, and data validation via CartDataInitializer, the login process will not be explained, everything is explained via comments in the repo and the request flow has been previously explained.

1. User makes a GET request to /menu.
2. Request gets redirected to the index.php because of the .htaccess configuration.

.htaccess

```
# Prevent direct access to sensitive files
<FilesMatch "\.(htaccess|htpasswd|ini|phps|fla|psd|log|sh|json|env)$">
    Order Allow,Deny
    Deny from all
</FilesMatch>

<IfModule mod_rewrite.c>
    RewriteEngine On

    # Exclude specific directories from rewriting
    RewriteRule ^(src/front/assets|css|scripts)/ - [L]

    # Rewrite all other URLs to index.php
    RewriteCond %{REQUEST_FILENAME} !-f
    RewriteCond %{REQUEST_FILENAME} !-d
    RewriteRule ^ index.php [L]
</IfModule>

```

Image 18. .htaccess configuration

3. Request reaches index.php and gets parsed.

index.php M

index.php > ...

```php
<?php

require_once __DIR__ . '/vendor/autoload.php';

//I am using the dotenv package to load the environment variables
$dotenv = Dotenv\Dotenv::createImmutable(__DIR__);
$dotenv->safeLoad();

//I am autoloading the classes with the composer autoload, so my custom one
//is not needed anymore
// require_once __DIR__ . '/src/autoload.php';

//Starting the session, this will make the $_SESSION variable available in the rest of
the application
session_start();

use App\View;
use App\AuthMiddleware;

define('ROOT_PATH', __DIR__);
define('VIEW_PATH', __DIR__ . '/src/front/views');
define('STORAGE_PATH', __DIR__ . '/src/front/assets/storage');

try {
    //Using AuthMiddleware class to check if the user is authenticated
    AuthMiddleware::checkAuth();

    require_once __DIR__ . '/src/lib/routes.php';
} catch (App\Exceptions\RouteNotFoundException | App\Exceptions\ViewNotFoundException $e)
{
    //Handling the exceptions
    if ($e instanceof App\Exceptions\RouteNotFoundException) {
        $statusCode = 404;
        $viewPath = 'error/404';
    } elseif ($e instanceof App\Exceptions\ViewNotFoundException) {
        $statusCode = 404;
        $viewPath = 'error/404';
    }

    //Setting the status code and rendering the view
    http_response_code($statusCode);
    echo (new View($viewPath))->render();
}
```

Image 19. index.php entry point

4. Request gets validated in the AuthMiddleware.

index.php M | AuthMiddleware.php

src > App > AuthMiddleware.php > PHP Intelephense > AuthMiddleware > checkAuth

```php
<?php

declare(strict_types=1);

namespace App;

use App\AuthSession;
use App\ResponseStatus;

/**
 * Class AuthMiddleware
 *
 * This class provides middleware functionality for authentication.
 */
2 references | 0 implementations
class AuthMiddleware
{
    /**
     * Checks if the user is authorized to access the requested route.
     * If not authorized, it sends a response with the appropriate status code and
     redirect URL.
     */
    1 reference | 0 overrides
    static public function checkAuth(): void
    {
        // Get the requested route
        $route = explode('?', $_SERVER['REQUEST_URI'])[0];

        // Get the role from the route
        $route_role = explode('/', $route)[1] ?? null;


        // Check if the user is authorized based on the route role
        if ($route_role === 'user' && !AuthSession::isLoggedIn()) {
            ResponseStatus::sendResponseStatus(403, 'You are not authorized', '/login');
        }

        if ($route_role === 'admin') {
            $userRole = AuthSession::getUserRole();
            if (!in_array($userRole, ['admin', 'owner'])) {
                ResponseStatus::sendResponseStatus(403, 'You are not authorized', '/');
            }
        }

        if ($route_role === 'owner') {
            if (!AuthSession::isUserRole('owner')) {
                ResponseStatus::sendResponseStatus(403, 'You are not authorized', '/');
            }
        }
    }
}
```

Image 20. AuthMiddleware

For the authentication 3 route keywords in the first route segment have been reserved, these will mark which user role is needed to access the route. The AuthMiddleware checkAuth() method checks if one of the 3 keywords is used, and if is used it checks if the person making the request is authorized to reach that route, if not, the execution of the whole script is stopped and status and response messages are sent to the client via ResponseStatus class. As the /menu is not a private route every user can visit it so the request continues ahead with the execution of the script.

The ResponseStatus class is a custom class created for handling responses to the client and stopping the script execution. The responses for Ajax requests and normal requests differ.

index.php M routes.php BaseController.php ResponseStatus.php M

src > App > ResponseStatus.php > PHP Intelephense > ResponseStatus > sendResponseStatus

```php
<?php

declare(strict_types=1);

namespace App;

110 references | 0 implementations
class ResponseStatus
{
    /**
     * Sends an error status response and stop script execution.
     *
     * @param int $status_code The HTTP status code to send.
     * @param string|null $message The message to send. Default is null.
     * @param string|null $redirect_url The URL to redirect to after sending the
    response. Default is null.
     * @param bool|null $is_ajax_response Indicates if the response is an AJAX response.
    Default is false.
     * @return void
     */
    99 references | 0 overrides
    static public function sendResponseStatus(int $status_code, ?string $message = null, ?
    string $redirect_url = null, ?bool $is_ajax_response = false): void
    {
        if ($is_ajax_response) {
            // Set the response code
            http_response_code($status_code);
            //Set the response content type
            if ($redirect_url) {
                //This one redirect the user to the URL
                echo json_encode($redirect_url);
            } else {

                echo json_encode($message);
            }
        } else {
            // Set the session message
            if ($message) {
                $status_code === 200 ? $_SESSION['success'] = $message : $_SESSION
                ['error'] = $message;
            }
            // Set the response code
            http_response_code($status_code);
            // Redirect if needed
            if ($redirect_url) {
                header('Location: ' . $redirect_url);
            }
        }

        exit();
    }
```

Image 21. ResponseStatus class

5. Request continues as it is not a protected route and the route declarations are reached.

index.php M routes.php

src > lib > routes.php > ...

```php
<?php

/**
 * This file contains the routes configuration for the application.
 * It defines the routes and their corresponding controller methods.
 */

$router = new \App\Router();

//Seed route
if ($_ENV['ENVIRONMENT'] === 'dev') {
    $router->get('/seed', [App\Seed\SeedController::class, 'seedDatabase']);
}

//Public routes
$router->get('/index.php', [App\Controllers\BaseController::class, 'getHomePage']);
$router->get('/', [App\Controllers\BaseController::class, 'getHomePage']);

$router->get('/menu', [App\Controllers\BaseController::class, 'getMenu']);
$router->get('/cart', [App\Controllers\CartController::class, 'getCart']);

$router->get('/login', [App\Controllers\AuthController::class, 'getLogIn']);
$router->post('/login', [App\Controllers\AuthController::class, 'postLogIn']);

$router->post('/register', [App\Controllers\AuthController::class, 'postRegister']);

$router->get('/logout', [App\Controllers\AuthController::class, 'getLogOut']);

//Protected routes

//User routes
$router->get('/user/account', [App\Controllers\User\UserController::class,
'getUserAccount']);
$router->post('/user/account', [App\Controllers\User\UserController::class,
'updateUserAccount']);

//Orders routes
$router->post('/user/order', [App\Controllers\OrderController::class, 'postOrderAjax']);
$router->get('/user/orders', [App\Controllers\OrderController::class, 'getUserOrders']);
$router->get('/user/order', [App\Controllers\OrderController::class, 'getUserOrder']);
$router->get('/user/order/cancel', [App\Controllers\OrderController::class,
'cancelPendingOrder']);


//Admin routes
$router->get('/admin/orders', [App\Controllers\Admin\AdminOrderController::class,
'getOrders']);
$router->get('/admin/order', [App\Controllers\Admin\AdminOrderController::class,
```

Image 22. Route registering

index.php M routes.php

src > lib > routes.php > ...

```php

//Admin routes
$router->get('/admin/orders', [App\Controllers\Admin\AdminOrderController::class, 'getOrders']);
$router->get('/admin/order', [App\Controllers\Admin\AdminOrderController::class, 'getOrder']);
$router->post('/admin/order/status', [App\Controllers\Admin\AdminOrderController::class, 'changeOrderStatus']);

$router->get('/admin/menu', [App\Controllers\Admin\AdminMenuController::class, 'getMenu']);

$router->get('/admin/product/new', [App\Controllers\Admin\AdminMenuController::class, 'getNewProduct']);
$router->post('/admin/product/new', [App\Controllers\Admin\AdminMenuController::class, 'postNewProductAjax']);

$router->get('/admin/product/update', [App\Controllers\Admin\AdminMenuController::class, 'getUpdateProduct']);
$router->post('/admin/product/update', [App\Controllers\Admin\AdminMenuController::class, 'postUpdateProductAjax']);
$router->delete('/admin/product/delete', [App\Controllers\Admin\AdminMenuController::class, 'deleteProductAjax']);

//Owner routes
$router->get('/owner/users', [App\Controllers\Owner\OwnerUserController::class, 'getUsers']);
$router->get('/owner/user/search', [App\Controllers\Owner\OwnerUserController::class, 'searchUser']);
$router->post('/owner/user/role', [App\Controllers\Owner\OwnerUserManagementController::class, 'updateUserRole']);


$router->get('/owner/user/management', [App\Controllers\Owner\OwnerUserManagementController::class, 'getManagementLogs']);
$router->get('/owner/user/management/search', [App\Controllers\Owner\OwnerUserManagementController::class, 'getLogsByUserEmail']);

//This is the line that is going to resolve the route and send the result to the client
echo $router->resolve($_SERVER['REQUEST_URI'], strtolower($_SERVER['REQUEST_METHOD']));

```

Image 23. Route registering and resolve

This is the file where all the routes for our project are declared, it uses a custom class called Router which is in charge of the routing functionalities. First we register all routes, the function called registers the request method, then this function takes 2 arguments.

The first one contains the request URI that will resolve the route, and the second one is a callable represented as an array that contains two elements, the first one is the class and the second the method that should be called in this class when resolving the route.

At the end of this file the route is being resolved using the resolve function from the Router class which takes two params, the request URI and the request method. If the route is not found the Router class will throw a custom Exception. More explanations on how the Router class works can be found inside the Router class.

6. In our case for the /menu URI the getMenu in the BaseController class will be called.

index.php M routes.php BaseController.php

src > App > Controllers > BaseController.php > PHP Intelephense > BaseController > getHomePage

```php
declare(strict_types=1);

namespace App\Controllers;

use App\View;
use App\ServerErrorLog;
use App\Models\ProductModel;
use App\ResponseStatus;

/**
 * The BaseController class is responsible for handling the main functionalities of the
 application.
 */
3 references | 0 implementations
class BaseController
{

    /**
     * Retrieves the home page view.
     *
     * @return string The rendered home page view.
     */
    2 references | 0 overrides
    public function getHomePage(): string
    {
        // Render the home page view
        return (new View('home'))->render();
    }

    /**
     * Retrieves the menu view.
     *
     * @return string |null The rendered menu view.
     */
    1 reference | 0 overrides
    public function getMenu(): ?string
    {
        try {
            // Get all products from the database
            $products = ProductModel::getAllByType();

            if (empty($products)) {
                // If no products are found, return a 404 error
                ResponseStatus::sendResponseStatus(404, 'No products found.');
            }

            // Render the view with the products
            return (new View('menu', $products))->render();
        } catch (\Exception $e) {
            // Log the error and return a 500 error
            ServerErrorLog::logError($e);
            ResponseStatus::sendResponseStatus(500, 'Error while loading the menu. Please
            try again later.');
        }
    }
}
```

Image 24. Base controller

6.1 First we get the products ordered by type using the ProductModel::getAllByType function. If there are no products a not ok response will be sent back to the client. If there is a view with the products will be rendered.

```php
/**
 * Retrieves all products sorted by type.
 *
 * @return array An array containing the products sorted by type.
 */
2 references | 0 overrides
static public function getAllByType(): array
{
    // Get all products from the database
    $productDAO = new ProductDAO();
    $products = $productDAO->getAllProducts();

    $sortedByType = [];

    foreach (DISHES_TYPES as $type) {
        $typeProducts = array_filter($products, fn ($product) => $product->getType()
        === $type);
        $sortedByType[$type] = $typeProducts;
    }

    return $sortedByType;
}
```

Image 25. Product Model getAllByType method

6.2 The getAllByType function will create an instance of the ProductDAO (which connects to the database when an instance is created using a Singleton Pattern) and gets all products from our database.

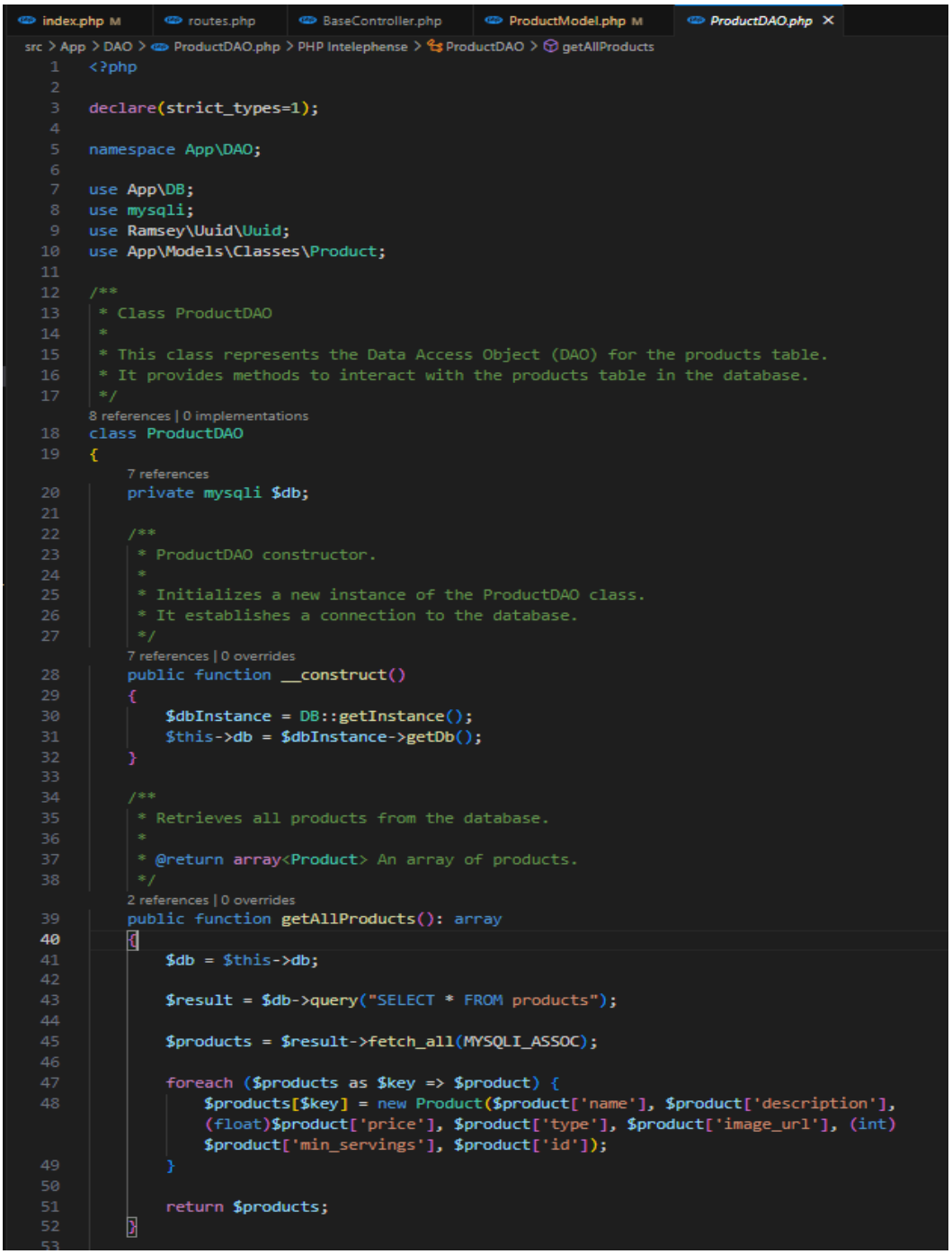


Image 26. Product DAO

7. If everything went okay the desired view will be rendered, in this case the menu.php view. This is possible thanks to the custom View class.

```
    // Render the view with the products
    return (new View('menu', $products))->render();
} catch (\Exception $e) {
```

Image 27. Render view products as passing params

```
use App\Exceptions\ViewNotFoundException;
use App\Models\Classes\CustomClass;

/**
 * Class View
 *
 * Represents a view in the application.
 */
class View
{
    /**
     * The path to the view file.
     *
     * @var string
     */
    private string $view;

    /**
     * The parameters to be passed to the view.
     *
     * @var array
     */
    private array | CustomClass $params;

    /**
     * Create a new View instance.
     *
     * @param string $view The path to the view file.
     * @param array|CustomClass $params The parameters to be passed to the view.
     */
    public function __construct(string $view, array|CustomClass $params = [])
    {
        $this->view = $view;
        $this->params = $params;
    }

    /**
     * Render the view and return the output as a string.
     *
     * @return string The rendered view.
     * @throws ViewNotFoundException If the view file does not exist.
     */
    public function render(): string
    {
        $viewPath = VIEW_PATH . '/' . $this->view . '.php';

        if (!file_exists($viewPath)) {
            throw new ViewNotFoundException();
        }

        // Extracting the parameters so they are available in the view with a variable
        variable

        $params = $this->params;

        /*  foreach ($this->params as $key => $value) {
            $$key = $value;
        } */

        // Buffering the output so it is returned as a string
        ob_start();
        include $viewPath;
        return (string) ob_get_clean();
    }
}
```

Image 28. View class

8. The menu view is rendered. I will not showcase here the php code for the view because it will make everything too long and if I have to display all the php code for the whole project this will never end.

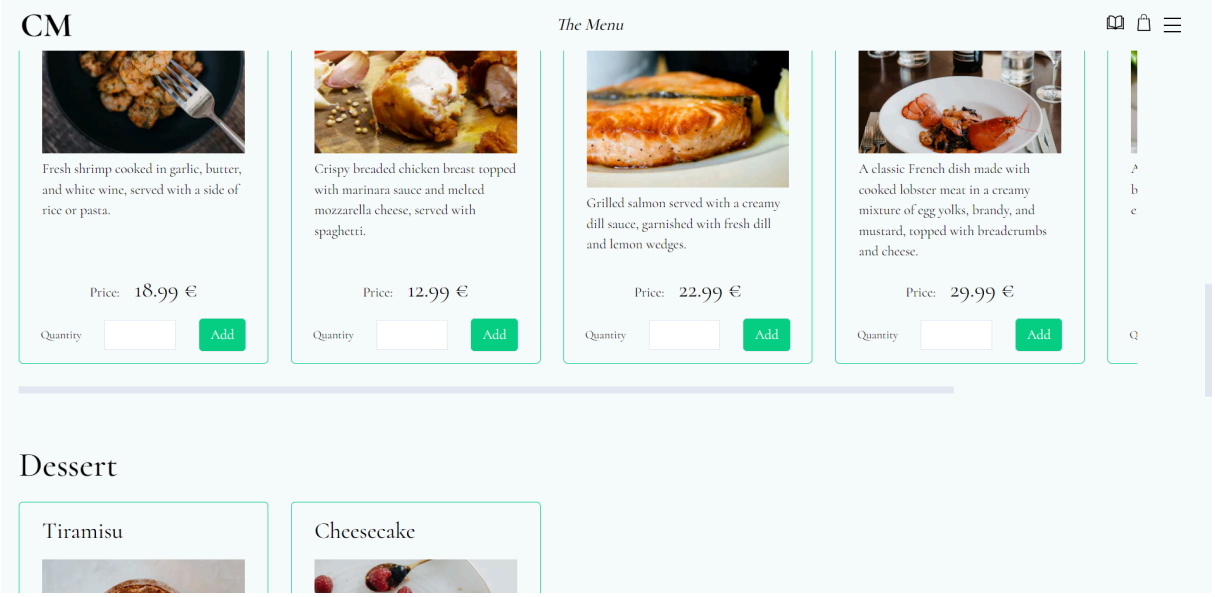


Image 29. Menu page

This page includes the following features:

- A x-axis draggable container for the lists when the quantity of the products overlay the width of the screen. This display makes it easier for the user to read between categories because if I had all displayed in a vertical grid there will be too many dishes and the customer could be overwhelmed, in the other hand how I implemented it the user always has enough information to visualize a lot of the categories of the menu and not get directly overwhelmed with all their items.
- When a click the button to add an item to the cart the following script being executed, adding the quantity to the cart if the minimum required quantity is reached:

add-to-cart.js

src > front > scripts > menu > add-to-cart.js > addForm.addEventListener('submit') callback

```
import { setCookie, getCookie } from './../cookies.js';

const addingToCartForms = document.querySelectorAll('.add-form');

for (const addForm of addingToCartForms) {
  addForm.addEventListener('submit', async (e) => {
    e.preventDefault();

    //parsing the form data
    const form = new FormData(addForm);

    const id = form.get('id');
    const minServings = parseInt(form.get('min_servings'));
    const quantity = parseInt(form.get('quantity'));
    try {
      // Get the cart from the cookie
      let cart = getCookie('cart') || [];
      //Checkin if the item already exists in the cart
      const existingItemIndex = cart.findIndex((item) => item.id === id);

      //If the item already exists in the cart, proceed to add the quantity to the existing quantity
      if (existingItemIndex !== -1) {
        //Check if the total quantity is less than the minimum servings, this helps in case a user tries to add a quantity lower than the minServings but that still reachs the minServings when added to the existing quantity
        if (quantity + cart[existingItemIndex].quantity < minServings) {
          alert('Minimum servings not reached, order more quantity to proceed.');
          return;
        }
        //If the total quantity is greater than the minimum servings, proceed to add the quantity to the existing quantity
        cart[existingItemIndex].quantity += quantity;
      } else {
        //If the item does not exist in the cart, check if the quantity is less than the minimum servings
        if (quantity < minServings) {
          alert('Minimum servings not reached, order more quantity to proceed.');
          return;
        }
        //In case there is no such item in the cart, add the item to the cart
        cart.push({ id, quantity });
      }

      //Save the cart to the cookie
      setCookie('cart', JSON.stringify(cart), 7);

      alert('Item added to cart successfully');
    } catch (error) {
      console.log(error);
      alert('An error occured while adding the item to your cart.');
    }
  });
}

```

Image 30. JavaScript add-to-cart.js script

For working with the cookies I created some functions:

routes.php | CartController.php | cookies.js M

src > front > scripts > cookies.js > getCookie

```
 * Sets a cookie with the specified name, value, and expiration days.
 * @param {string} cname - The name of the cookie.
 * @param {string} cvalue - The value of the cookie.
 * @param {number} [exdays=30] - The number of days until the cookie expires (default is 30 days).
 */
export function setCookie(cname, cvalue, exdays = 30) {
  // Create a new date object
  const d = new Date();
  // Set the expiration date
  d.setTime(d.getTime() + exdays * 24 * 60 * 60 * 1000);
  // Set the cookie
  let expires = 'expires=' + d.toUTCString();
  document.cookie = cname + '=' + cvalue + ';' + expires + ';path=/';
}

/**
 * Retrieves the value of a cookie by its name and return its value in JSON format.
 * @param {string} cname - The name of the cookie.
 * @returns {any} The value of the cookie in JSON format, or null if the cookie does not exist.
 */
export function getCookie(cname) {
  // Get the cookie by its name
  let name = cname + '=';
  let ca = document.cookie.split(';');
  // Loop through the cookies
  for (let i = 0; i < ca.length; i++) {
    // Get the current cookie
    let c = ca[i];
    // Remove leading spaces
    while (c.charAt(0) == ' ') {
      c = c.substring(1);
    }
    // Check if the cookie is the one we are looking for
    if (c.indexOf(name) == 0) {
      const decodedCookie = decodeURIComponent(c.substring(name.length, c.length));
      return JSON.parse(decodedCookie);
    }
  }
  return null;
}
```

Image 31. Cookie functions

9. Once the user has added the desired items to the cart he can click the shopping bag icon in the navigation bar in order to proceed to the cart page. This generates a request to the /cart url generating the same flow as I explained before, .htaccess, index.php, AuthMiddleware and route resolving.

The /cart url resolves to this class:

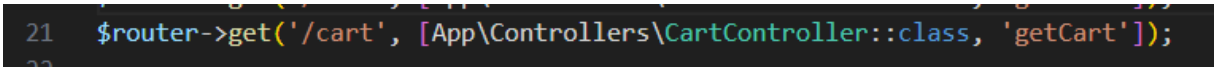


Image 32. getCart route

10. The getCart method from the CartController gets executed.

routes.php CartController.php

src > App > Controllers > CartController.php > PHP Intelephense > CartController

```php
<?php

declare(strict_types=1);

namespace App\Controllers;

use App\AuthSession;
use App\Models\Cart\CartDataInitializer;
use App\ResponseStatus;
use App\ServerErrorLog;
use App\View;

/**
 * The CartController class is responsible for handling the main cart functionalities of
 the application.
 */
1 reference | 0 implementations
class CartController
{

    /**
     * Retrieves the cart view.
     *
     * @return string The rendered cart view.
     */
    1 reference | 0 overrides
    public function getCart(): ?string
    {
        try {
            // Get the products from the cart
            $cart = new CartDataInitializer();
            $cartProducts = $cart->getProducts();
            $totalPrice = $cart->getTotalPrice();

            // Get the user name from the session
            $user_name = AuthSession::getUserName();

            // Render the view with the cart products and total price
            return (new View('cart', ['products' => $cartProducts, 'totalPrice' =>
            $totalPrice, 'user_name' => $user_name]))->render();
        } catch (\Exception $e) {
            // Log the error and return a 500 error
            ServerErrorLog::logError($e);
            ResponseStatus::sendResponseStatus(500, 'Error while loading the cart. Please
            try again later.');
        }
    }
}
```

Image 33. getCart function in CartController

The CartDataInitializer is a pretty important class because it is in charge of cleaning up and merging the database products with the cart cookie products, eliminating from the cookie the items that are no longer available and updating the minimum required quantity when needed. This class allows us to maintain the integrity of the data.

routes.php | CartController.php | CartDataInitializer.php

src > App > Models > Cart > CartDataInitializer.php > PHP Intelephense > CartDataInitializer > __construct

```php
<?php

declare(strict_types=1);

namespace App\Models\Cart;

use App\Models\ProductModel;
use App\Models\Classes\Product;
use App\ResponseStatus;

/**
 * Class CartDataInitializer
 * Prepares the cart the complete cart data, including merging cookie data with fetched products and calculating totals.
 * For working with the cart cookie use CartCookie.
 * For making database operation use  CartDb .
 */
10 references | 0 implementations
class CartDataInitializer
{

    /**
     * @var CartProductData[] $products An array of CartProductData objects representing the products in the cart.
     * @var float $totalPrice The total price of all products in the cart.
     */
    4 references
    private array $products = [];
    2 references
    private float $totalPrice = 0;

    /**
     * CartDataInitializer constructor.
     * Initializes the cart data (primarily for display) by merging fetched products with cookie cart items and calculating the total price.
     */
    2 references | 0 overrides
    public function __construct()
    {
        $this->initializeCartData();
        $this->calculateTotalPrice();
    }

    /**
     * Initializes the products in the cart by merging fetched products with cookie cart items.
     */
    1 reference | 0 overrides
    public function initializeCartData(): void
```

Image 34. CartDataInitalizer class

```php
/**
 * Initializes the products in the cart by merging fetched products with cookie cart
 items.
 */
1 reference | 0 overrides
public function initializeCartData(): void
{
    // Get the cart from the cookie
    $cookieCart = new CartCookie();
    $cookieCartData = $cookieCart->getCart();

    // Get the ids of the products in the cart
    $id_list = $cookieCart->getIds();

    // If the cart is empty, redirect to the menu page
    //I know that this is not the best way to handle this, but I don't have time to
    implement a better solution
    if (count($id_list) === 0) {
        ResponseStatus::sendResponseStatus(500, 'Your cart is empty, add some items
        to it first.', '/menu');
    }

    // Get the products from the database
    $result = ProductModel::getProductsByIds($id_list);

    $products = $result['products'];
    $productNotFound = $result['not_found'];

    $mergedProducts = $this->mergeProducts($products, $cookieCartData);

    $this->products = $mergedProducts;

    if ($productNotFound) {
        $cartCookie = new CartCookie($this);
        $cartCookie->saveCart();

        $_SESSION['error'] = 'Some of your cart products were modified or deleted,
        your cart has been updated.';
    }
}
```

Image 35. initializeCart method in CartDataInitializer class

```php
    /**
     * Merges fetched products with cookie cart items.
     *
     * @param Product[] $fetchedProducts The fetched products from the database.
     * @param array $cookieCartData The cart items from the cookie.
     * @return CartProductData[] The merged products.
     */
    1 reference
    private function mergeProducts(array $fetchedProducts, array $cookieCartData): array
    {
        $products = [];

        foreach ($fetchedProducts as $product) {
            $found = false;
            $index = 0;
            do {
                $cookieItem = $cookieCartData[$index];
                $cookieId =  $cookieItem['id'];
                $productId =  $product->getId();


                if ($cookieId === $productId) {

                    // If the quantity of the product in the cart is less than the
                    minimum servings, set it to the minimum servings
                    if ($product->getMinServings() > $cookieItem['quantity']) {
                        $cookieItem['quantity'] = $product->getMinServings();
                    }

                    $products[] = new CartProductData($productId, $product->getName(),
                    $cookieItem['quantity'], $product->getPrice(), $product->getType(),
                    $product->getImageUrl(), $product->getDescription(),
                    $product->getMinServings());
                    $found = true;
                }

                $index++;
            } while (!$found && $index < count($cookieCartData));
        }

        return $products;
    }
```

Image 36. mergeProducts method in CartDataInitializer class

```php
    /**
     * Calculates the total price of all products in the cart.
     */
    1 reference | 0 overrides
    public function calculateTotalPrice(): void
    {
        $totalPrice = 0;

        foreach ($this->products as $product) {
            $totalPrice += $product->getPrice() * $product->getQuantity();
        }

        $this->totalPrice = (float) number_format($totalPrice, 2, '.', '');
    }

    /**
     * Retrieves the products in the cart.
     *
     * @return array The products in the cart.
     */
    2 references | 0 overrides
    public function getProducts(): array
    {
        return $this->products;
    }

    /**
     * Retrieves the total price of all products in the cart.
     *
     * @return float The total price of all products in the cart.
     */
    2 references | 0 overrides
    public function getTotalPrice(): float
    {
        return $this->totalPrice;
    }

    /**
     * Generates an array of data objects representing the products in the cart.
     *
     * @return array The array of data objects representing the products in the cart.
     */
    1 reference | 0 overrides
    public function generateProductsDataObject(): array
    {
        $cartData = [];

        foreach ($this->products as $product) {
            $cartData[] = $product->generateDataObject();
        }

        return $cartData;
    }
}
```

Image 37. Rest of methods in CartDataInitializer class

11. If no products are in the cart the customer will be redirected to the /menu page and display a message telling him that he has no products in the cart.

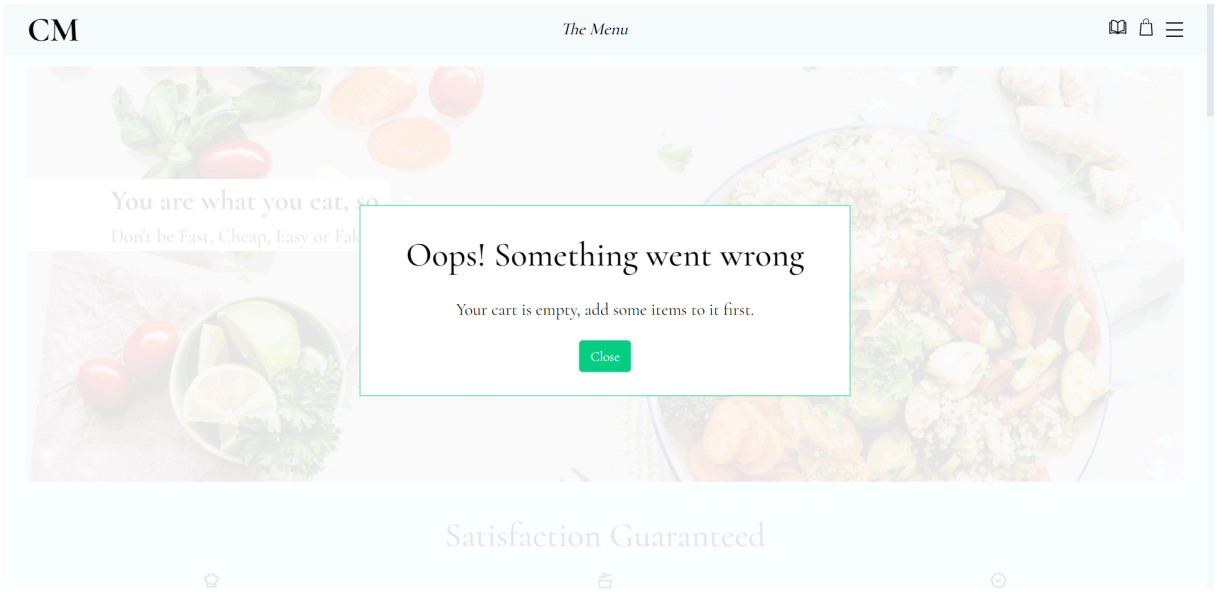


Image 38. Error message display

12. If he has products in the cart the View is rendered.

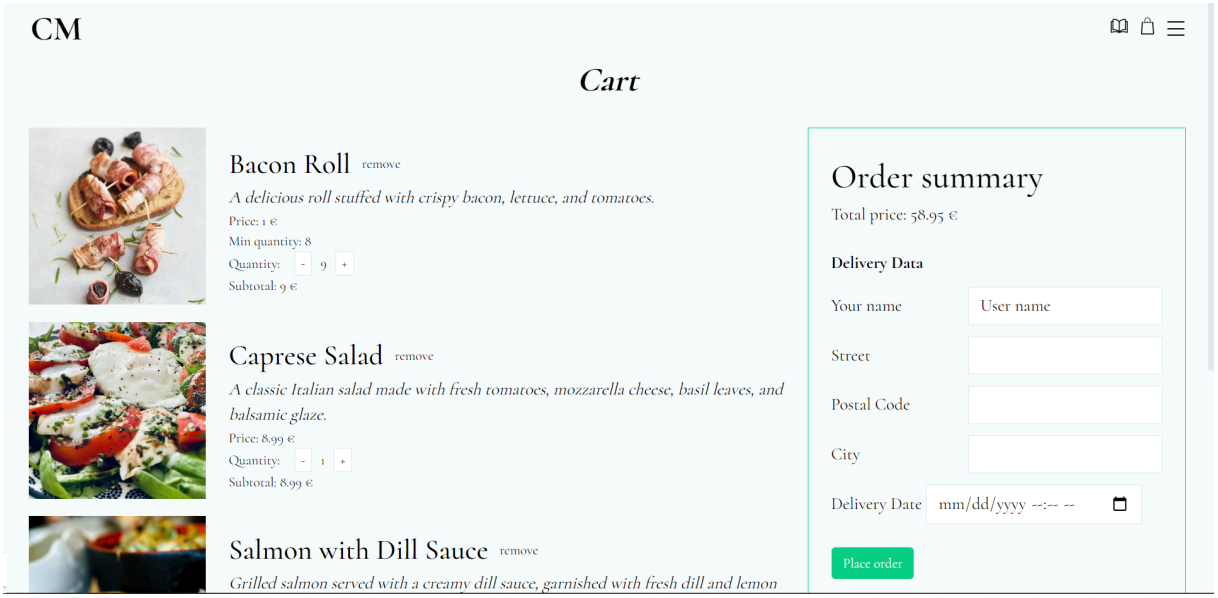


Image 39. Cart page

This page includes the following features:

- Event listeners for - , + and remove buttons that modify the cookie and modify the values of the prices in the client after the modifications.

routes.php | add-one.js

src > front > scripts > cart > add-one.js > ...

```
import { getCookie, setCookie } from '../cookies.js';

const addButtons = document.querySelectorAll('.add-button');

for (const addButton of addButtons) {
  addButton.addEventListener('click', (e) => {
    e.preventDefault();
    const itemLi = addButton.closest('.cart-item-li');

    const productId = addButton.dataset.productid;
    let initialQuantity = parseInt(itemLi.querySelector('.quantity-span').innerHTML);
    const itemPrice = parseFloat(addButton.dataset.productprice);
    const newQuantity = ++initialQuantity;
    const newSubtotal = newQuantity * itemPrice;
    const newTotalPrice = parseFloat(document.querySelector('.total-price').innerHTML) +
    itemPrice;

    try {
      //Getting the cookie, updating the product quantity and saving the new cart
      const cart = getCookie('cart');

      const newCart = cart.map((item) => {
        if (item.id === productId) {
          item.quantity = newQuantity;
        }
        return item;
      });
      console.log(newCart);
      setCookie('cart', JSON.stringify(newCart));

      //Changing the output data in the html file
      itemLi.querySelector('.quantity-span').innerHTML = newQuantity;
      itemLi.querySelector('.item-subtotal-price').innerHTML = newSubtotal.toFixed(2);
      document.querySelector('.total-price').innerHTML = newTotalPrice.toFixed(2);
    } catch (error) {
      console.log(error);
      return;
    }
  });
}
```

Image 40. add-one.js script

add-one.js | remove-from-cart.js | remove-one.js

src > front > scripts > cart > remove-one.js > button.addEventListener('click') callback > newCart > cart.map() callback

```
import { getCookie, setCookie } from '../cookies.js';

const removeButtons = document.querySelectorAll('.reduce-button');

for (const button of removeButtons) {
  button.addEventListener('click', (e) => {
    e.preventDefault();

    const itemLi = button.closest('.cart-item-li');

    const productId = button.dataset.productid;
    let initialQuantity = parseInt(itemLi.querySelector('.quantity-span').innerHTML);
    const itemPrice = parseFloat(button.dataset.productprice);
    const minServings = parseInt(button.dataset.minservings);
    const newQuantity = --initialQuantity;
    const newSubtotal = newQuantity * itemPrice;
    const newTotalPrice = parseFloat(document.querySelector('.total-price').innerHTML) -
    itemPrice;
    console.log(minServings);
    if (newQuantity < minServings || newQuantity < 1) {
      alert('Minimum required quantity required.');
      return;
    }

    try {
      //Getting the cookie, updating the product quantity and saving the new cart
      const cart = getCookie('cart');
      const newCart = cart.map((item) => {
        if (item.id === productId) {
          item.quantity = newQuantity;
        }
        return item;
      });
      setCookie('cart', JSON.stringify(newCart));

      //Changing the output data in the html file
      itemLi.querySelector('.quantity-span').innerHTML = newQuantity;
      itemLi.querySelector('.item-subtotal-price').innerHTML = newSubtotal.toFixed(2);
      document.querySelector('.total-price').innerHTML = newTotalPrice.toFixed(2);
    } catch (error) {
      console.log('An error ocurred');
      return;
    }
  });
}

```

Image 40. remove-one.js script

add-one.js | remove-from-cart.js

src > front > scripts > cart > remove-from-cart.js > ...

```
1  import { getCookie, setCookie } from '../cookies.js';
2
3  const removeButtons = document.querySelectorAll('.remove-button');
4
5  for (const button of removeButtons) {
6    button.addEventListener('click', (e) => {
7      //Getting the product id and the subtotal price of the item to be removed
8      const productId = e.target.dataset.productid;
9      const subtotal = parseFloat(
10       button.closest('.cart-item-li').querySelector('.item-subtotal-price').innerHTML
11     ).toFixed(2);
12
13     //Getting the total price of the cart
14     const totalPriceElement = document.querySelector('.total-price');
15     const totalPrice = parseFloat(totalPriceElement.innerHTML).toFixed(2);
16
17     try {
18       //Getting the cookie, removing the product and saving the new cart
19       const cart = getCookie('cart');
20
21       const newCart = cart.filter((item) => item.id !== productId);
22
23       setCookie('cart', JSON.stringify(newCart));
24       // Hiding the item from the cart and updating the total price
25       button.closest('.cart-item-li').style.display = 'none';
26       totalPriceElement.innerHTML = (totalPrice - subtotal).toFixed(2);
27     } catch (e) {
28       console.log('An error occurred while removing the item from the cart.');
29       return;
30     }
31   });
32 }
33
```

Image 42.remove-from-cart.js script

- An AJAX request in order to create the order. When the data is given and the 'Order Summary' form is submitted an AJAX request is fired in order to create the order.

create-order.js

src > front > scripts > cart > create-order.js > ...

```
const form = document.getElementById('create-order-form');

if (form) {
  form.addEventListener('submit', async (e) => {
    e.preventDefault();

    const formData = new FormData(form);
    let response;

    try {
      response = await fetch('/user/order', {
        method: 'POST',
        body: formData,
      });
    } catch (error) {
      alert('Something went wrong, please try again later');
      return;
    }

    const responseData = await response.json();

    if (!response.ok) {
      alert(responseData || 'Something went wrong, please try again later');
      return;
    }

    window.location.href = responseData;
  });
}

```

Image 43. AJAX request

When the response is okay, the user will be redirected to the given URL given by the response data.

13. The POST request resolved in the following method and class:

```
//Orders routes
$router->post('/user/order', [App\Controllers\OrderController::class, 'postOrderAjax']);
```

Image 44. AJAX request route

OrderController.php | AddressModel.php | AddressDAO.php M

src > App > Controllers > OrderController.php > PHP Intelephense > OrderController > postOrderAjax

```
23  class OrderController
162     }
163
164     /**
165      * Handles the AJAX request to place an order.
166      *
167      * @return void
168      */
        1 reference | 0 overrides
169     public function postOrderAjax(): void
170     {
171         // Get the user ID from the session
172         $userId = AuthSession::getUserId();
173
174         if (!$userId) {
175             // If the user is not logged in, return a 401 error
176             ResponseStatus::sendResponseStatus(401, 'You must be logged in to place an
                order.', '/login', true);
177         }
178
179         // Get the form data
180         $user_name = trim($_POST['user_name']);
181         $street = trim($_POST['street']);
182         $postal = trim($_POST['postal']);
183         $city = trim($_POST['city']);
184         $delivery_date = $_POST['delivery_date'];
185
186         // Get the cart data from the cookie
187         $order_data = CartCookie::getCartFromCookie();
188
189
190         //Validate inputs
191         if ($delivery_date < date('Y-m-d H:i:s')) {
192             ResponseStatus::sendResponseStatus(400, 'Invalid date.', null, true);
193         }
194
195         if (!$order_data) {
196             // If there is no order data, return a 400 error
197             ResponseStatus::sendResponseStatus(400, 'No order data.', null, true);
198         }
199
200         // Check if the delivery data is invalid
201         $isInvalidData = isInvalidDeliveryData($user_name, $street, $postal, $city,
            $delivery_date);
202
203         if ($isInvalidData) {
204             // If the delivery data is invalid, return a 400 error
```

Image 45. postOrderAjax controller first part

OrderController.php | AddressModel.php | AddressDAO.php M

src > App > Controllers > OrderController.php > PHP Intelephense > OrderController > postOrderAjax

```
23  class OrderController
169     public function postOrderAjax(): void
198         }
199
200         // Check if the delivery data is invalid
201         $isInvalidData = isInvalidDeliveryData($user_name, $street, $postal, $city,
            $delivery_date);
202
203         if ($isInvalidData) {
204             // If the delivery data is invalid, return a 400 error
205             ResponseStatus::sendResponseStatus(400, $isInvalidData, null, true);
206         }
207
208         try {
209             // Save the address data
210             $addressModel = new AddressModel($street, $city, $postal,);
211             $addressId = $addressModel->saveAddressData();
212
213             if (!$addressId) {
214                 // If the address could not be saved, return a 500 error
215                 ResponseStatus::sendResponseStatus(500, 'An error ocurred. Try again
                    later.', null, true);
216             }
217
218             // Save the order data
219             $orderModel = new OrderModel($userId, $user_name, $addressId, new
                CartDataInitializer(), $delivery_date);
220             $success = $orderModel->saveOrderData();
221
222             if (!$success) {
223                 // If the order could not be saved, return a 500 error and delete the
                    stored address
224                 $addressModel->deleteAddressData($addressId);
225                 ResponseStatus::sendResponseStatus(500, 'An error ocurred. Try again
                    later.', null, true);
226             }
227
228             // Clear the cart cookie
229             CartCookie::destroyCartCookie();
230             // Redirect to the orders page
231             ResponseStatus::sendResponseStatus(200, null, '/user/orders', true);
232         } catch (\Exception $e) {
233             // Log the error and return a 500 error
234             ServerErrorLog::logError($e);
235             ResponseStatus::sendResponseStatus(500, 'An error ocurred. Try again later.',
                null, true);
236         }
237     }
238 }
```

Image 46. postOrderAjax controller second part

And after this, if everything went alright, an order will be created and the user will be redirected to /user/orders. I will not explain this code step by step as it is pretty self explanatory and all the code is commented in the repository, so if you want to know how everyone of these methods work you can check directly in the code and follow the comments.

5. Testing

Testing is the process of evaluating and verifying that the software designed performs as intended. There are two main categories when talking about software testing:

- Functional testing: Its purpose is to avoid code breaking when introducing changes to the codebase. They verify that everything is implemented as designed and if not the tests will not pass, pointing out where the problem was introduced. They can also mock functions and database connections so all the functionalities can be tested without making real requests to our database.
- Non functional testing: This test's purpose is to make some checks in order to metric performance, usability, security…

By combining both types of test we will have a pretty robust codebase where it is pretty difficult to introduce bugs or features that break how the code works as well as measuring different performance and critical metrics in our app, making it easier to improve and scale.

For this app I created 14 functional tests for the Router and View class, for which the PHPUnit testing library is being used.

tests > Unit > RouterTest.php > RouterTest > test_it_registers_a_route

```php
<?php

declare(strict_types=1);

namespace Tests\Unit;

use App\Router;
use PHPUnit\Framework\TestCase;
use App\Exceptions\RouteNotFoundException;
use PHPUnit\Framework\Attributes\DataProvider;

0 references | 0 implementations
class RouterTest extends TestCase
{

    22 references
    protected Router $router;

    1 reference | 0 overrides
    protected function setUp(): void
    {
        parent::setUp();

        $this->router = new Router();
    }
    0 references | 0 overrides
    public function test_it_registers_a_route(): void
    {
        //when we register a route
        $this->router->register('get', '/home', ['UserController', 'index']);

        //then the route should be stored in the routes array
        $expected = ['get' => ['/home' => ['UserController', 'index']]];
        $this->assertEquals($expected, $this->router->routes());
    }

    0 references | 0 overrides
    public function test_registers_get_route(): void
    {
        //given that we have router object
        $this->router = new Router();

        //when we register a GET route
        $this->router->get('/home', ['UserController', 'index']);

        //then the route should be stored in the routes array
        $expected = ['get' => ['/home' => ['UserController', 'index']]];
```

PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS COMMENTS

Image 47. Unit tests with PHPUnit

This is a part of the Router tests, I will discuss the first one to showcase how testing works. With the help of the PHPUnit library we have access to some testing functions, the test should be named after what it actually does, so it has more meaning when an error is being thrown when running the test.

For example, in the test_it_registers_a_route we know directly what this test tests just with the function name. Getting deeper into the function we can see that first we register a route with the Router register function.

Then we create a mock of what result we are expecting given the data we registered. With the help of the assertEquals PHPUnit method we can check if both the expected result and the actual result are the same.

If they are the same the code is working as intended, if for some reason we change something in the View class and no longer outputs what we expected a failed test will be thrown for this function and we would know that something is wrong when registering the route.

```
Runtime:       PHP 8.2.12
Configuration: C:\xampp\htdocs\proyecto-integrador\phpunit.xml

.....E.E                                                          14 / 14 (100%)

Time: 00:00.025, Memory: 8.00 MB

There were 2 errors:

1) Tests\Unit\ViewTest::test_it_renders_view
Error: Undefined constant "ROOT_PATH"

C:\xampp\htdocs\proyecto-integrador\src\front\views\includes\shared-head.php:8
C:\xampp\htdocs\proyecto-integrador\src\front\views\login.php:1
C:\xampp\htdocs\proyecto-integrador\src\App\View.php:68
C:\xampp\htdocs\proyecto-integrador\tests\Unit\ViewTest.php:21

2) Tests\Unit\ViewTest::test_params_is_accestible_in_view
Error: Call to undefined function generateSEOTags()

C:\xampp\htdocs\proyecto-integrador\src\front\views\testing.php:6
C:\xampp\htdocs\proyecto-integrador\src\App\View.php:68
C:\xampp\htdocs\proyecto-integrador\tests\Unit\ViewTest.php:43

--

There were 2 risky tests:

1) Tests\Unit\ViewTest::test_it_renders_view
Test code or tested code did not close its own output buffers

C:\xampp\htdocs\proyecto-integrador\tests\Unit\ViewTest.php:15

2) Tests\Unit\ViewTest::test_params_is_accestible_in_view
Test code or tested code did not close its own output buffers

C:\xampp\htdocs\proyecto-integrador\tests\Unit\ViewTest.php:39

ERRORS!
Tests: 14, Assertions: 12, Errors: 2, Risky: 2.
```

Image 48. Failed tests

This is the actual result of running tests and getting some error when passing the tests.

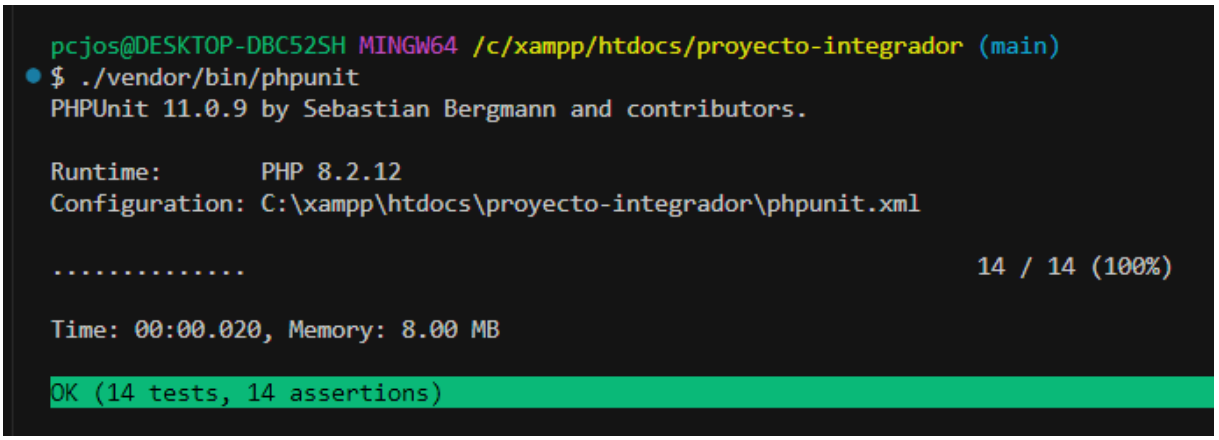


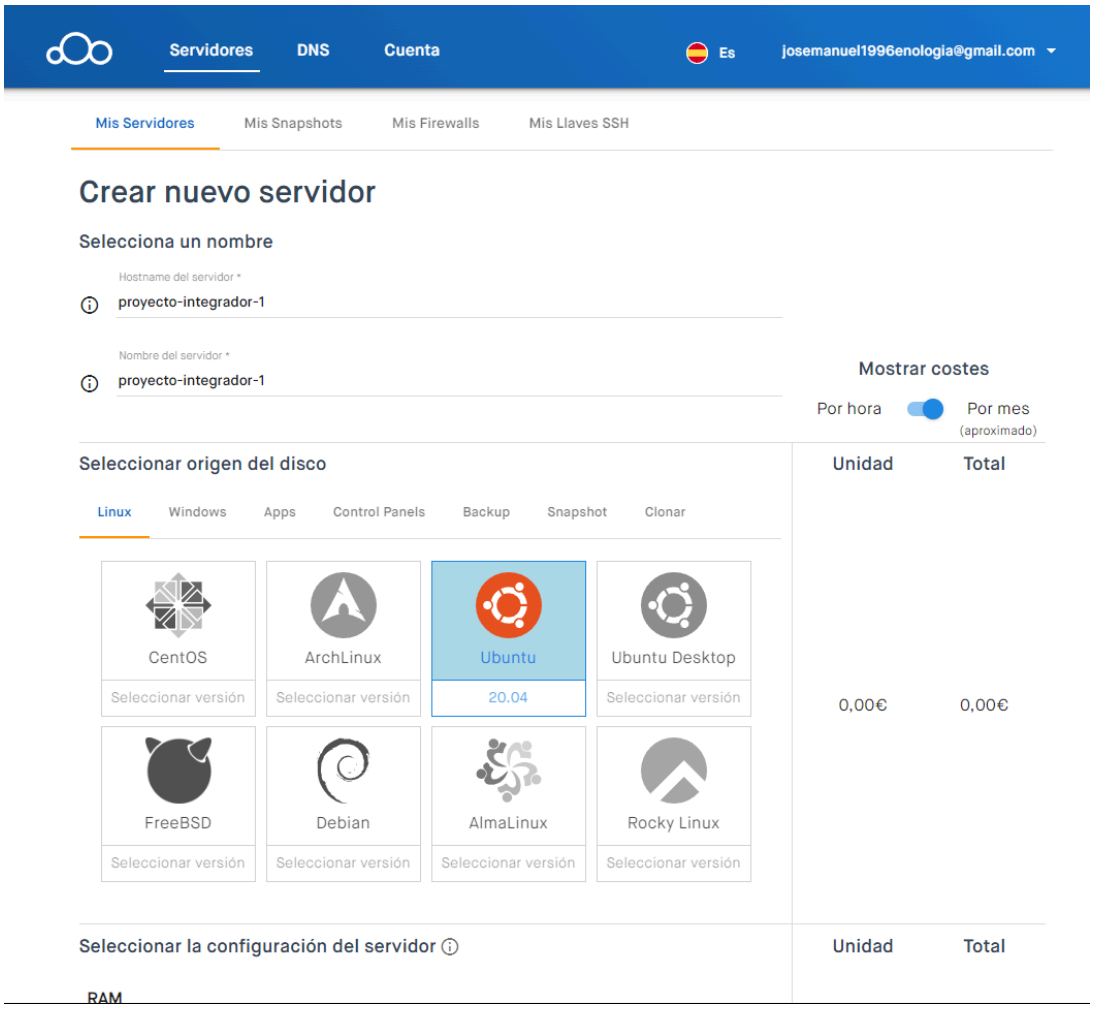
Image 49. Successful

This is the result after fixing the problems and passing all tests. There are 14 tests for this app, I didn't test everything because it was not the goal of the project and decided just to showcase how the test works for didactical purposes.

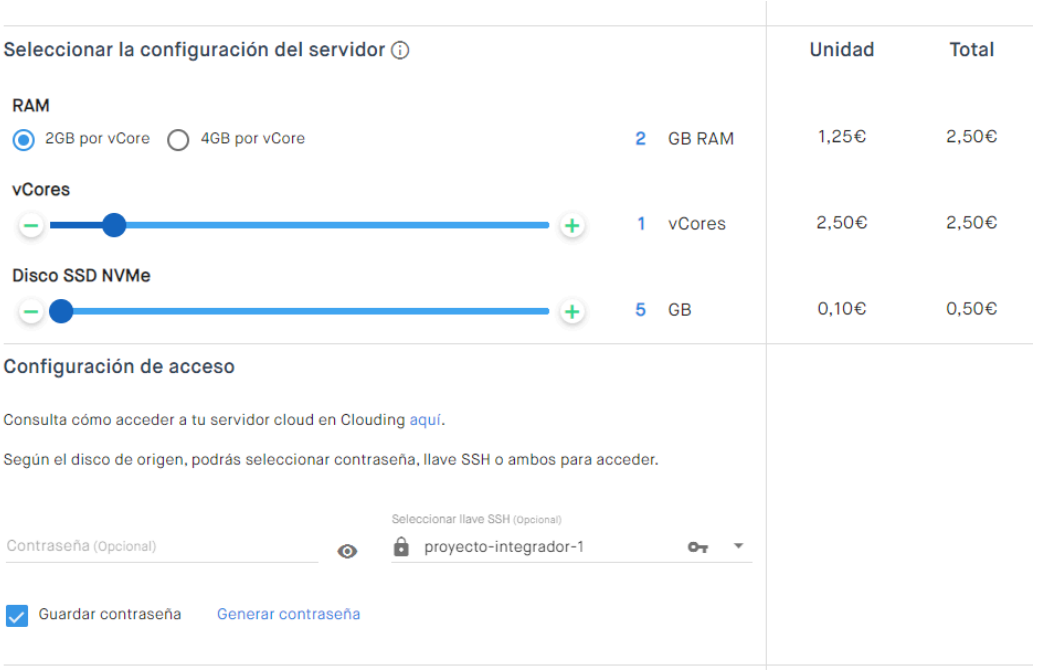
6. Deployment

The webpage has been deployed to a VPS in clouding.io using a LAMP stack. This stack has been completely installed and the server has been completely configured via command line connecting via SSH with PuTTy. The data for accessing the webpage is at the top of the document in the Demo section but you can visit the webpage in the following address http://161.22.40.113/ . In this section I will explain how this has been achieved:

1. Creating the VPS server with Ubuntu 20.04 installed in clouding.io.



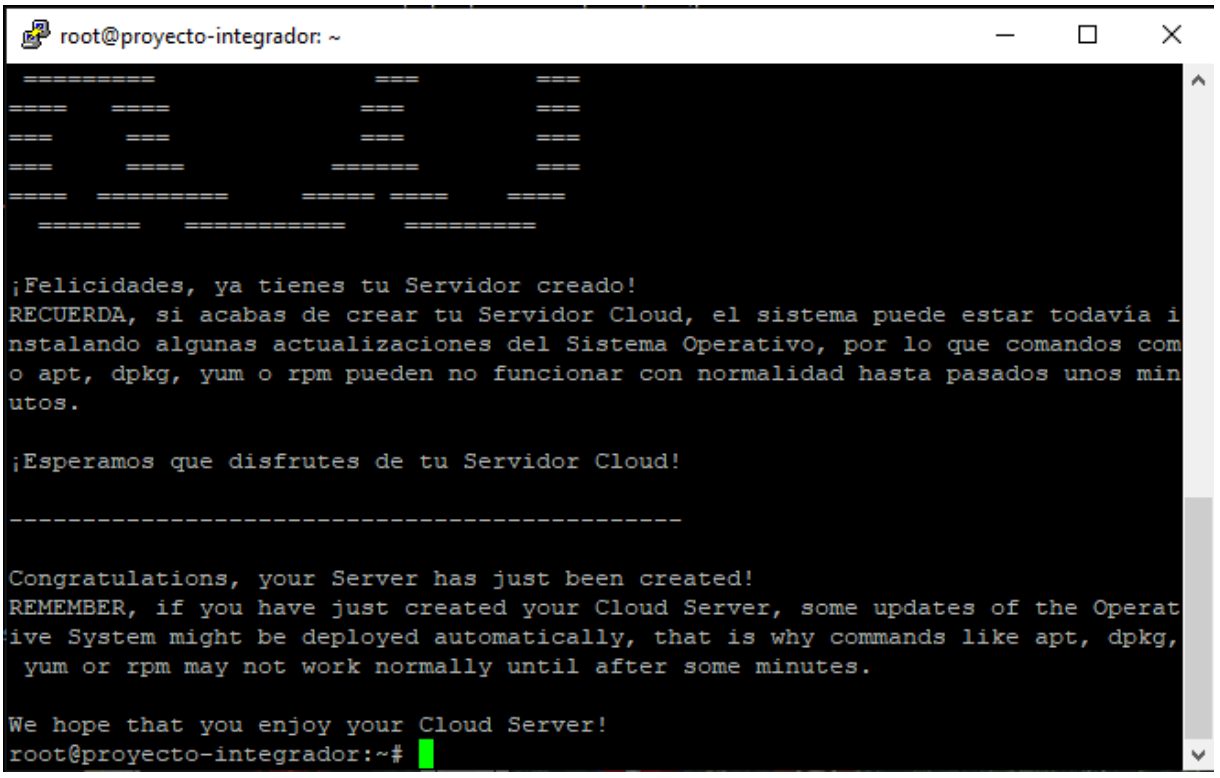
2. Selecting the server specs and creating an SSH key for accessing the project via command line later on and a password.



3. Once the server is up and running I connected via SSH key to it, for this, I followed the official clouding.io manual: https://help.clouding.io/hc/es/articles/360013896119-Primeros-pasos-con-mi-servidor-Linux

4. We will be using PuTTY to connect via SSH. You can check how to install and how it works here: https://help.clouding.io/hc/es/articles/360009983040-C%C3%B3mo-acceder-por-SSH-a-Linux-con-PuTTY

5. Once connected we can start installing all the things we need in our server. This is the terminal after you have successfully connected to your server using PuTTY.



6. First thing we do is we want to check if our server is up to date, for this we use the command ```apt update```. This command refreshes the list of available packages and their versions.

```
We hope that you enjoy your Cloud Server!
root@proyecto-integrador:~# apt update
```

7. Now we actually update all the packages running the command ```apt upgrade```, this command upgrades all the packages to the latest version.

```
root@proyecto-integrador:~# apt upgrade
Reading package lists... Done
```

8. Now that all packages are up to date we will install our Apache server, for this we run the command ```apt install apache2```.

```
root@proyecto-integrador:~# apt install apache2
Reading package lists... Done
```

9. Now we will install the MySQL server, for this we will run the ```apt install mysql-server```.

```
root@proyecto-integrador:~# apt install mysql-server
```

10. Now we will make some security changes to mysql-server, for this we will run the command ```mysql_secure_installation```. This will change the configuration in order to achieve a more secure mysql server.

```
root@proyecto-integrador:~# mysql_secure_installation
```

We chose this security configuration which is the most restrictive one:

```
Securing the MySQL server deployment.

Connecting to MySQL using a blank password.

VALIDATE PASSWORD COMPONENT can be used to test passwords
and improve security. It checks the strength of password
and allows the users to set only those passwords which are
secure enough. Would you like to setup VALIDATE PASSWORD component?

Press y|Y for Yes, any other key for No: y

There are three levels of password validation policy:

LOW    Length >= 8
MEDIUM Length >= 8, numeric, mixed case, and special characters
STRONG Length >= 8, numeric, mixed case, special characters and dictionary
            file

Please enter 0 = LOW, 1 = MEDIUM and 2 = STRONG: 2

Skipping password set for root as authentication with auth_socket is used by def
ault.
If you would like to use password authentication instead, this can be done with
the "ALTER_USER" command.
See https://dev.mysql.com/doc/refman/8.0/en/alter-user.html#alter-user-password-
management for more information.

By default, a MySQL installation has an anonymous user,
allowing anyone to log into MySQL without having to have
a user account created for them. This is intended only for
testing, and to make the installation go a bit smoother.
You should remove them before moving into a production
environment.

Remove anonymous users? (Press y|Y for Yes, any other key for No) : y
Success.
```

```
Remove anonymous users? (Press y|Y for Yes, any other key for No) : y
Success.


Normally, root should only be allowed to connect from
'localhost'. This ensures that someone cannot guess at
the root password from the network.

Disallow root login remotely? (Press y|Y for Yes, any other key for No) : y
Success.

By default, MySQL comes with a database named 'test' that
anyone can access. This is also intended only for testing,
and should be removed before moving into a production
environment.


Remove test database and access to it? (Press y|Y for Yes, any other key for No)
 : y
 - Dropping test database...
Success.

 - Removing privileges on test database...
Success.

Reloading the privilege tables will ensure that all changes
made so far will take effect immediately.

Reload privilege tables now? (Press y|Y for Yes, any other key for No) : y
Success.

All done!
root@proyecto-integrador:~#
```

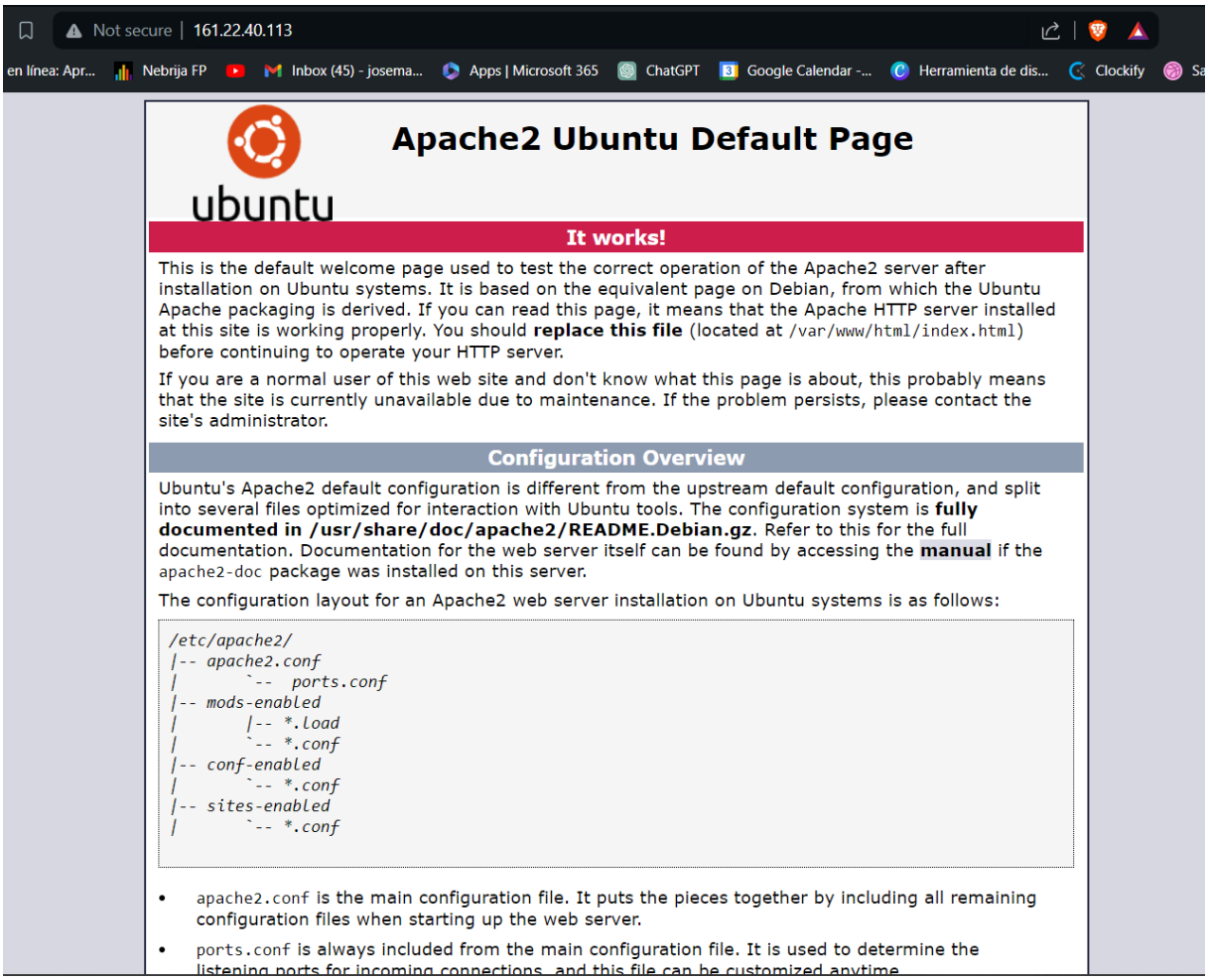
11. Note that we are getting authenticated in MySQL via an auth_socket, so no password is needed for the MySQL user as soon as the user authenticated in the localhost from the server is the same from the MySQL server, we will change that later.
12. As a next step we will install PHP using the following command ```apt install php libapache2-mod-php php-mysql```.

```
root@proyecto-integrador:~# apt install php libapache2-mod-php php-mysql
```

13. We use the command ```php -v``` to check that we have PHP correctly installed, this command will return the PHP version installed.

```
root@proyecto-integrador:~# php -v
PHP 7.4.3-4ubuntu2.22 (cli) (built: May  1 2024 10:11:33) ( NTS )
Copyright (c) The PHP Group
Zend Engine v3.4.0, Copyright (c) Zend Technologies
    with Zend OPcache v7.4.3-4ubuntu2.22, Copyright (c), by Zend Technologies
```

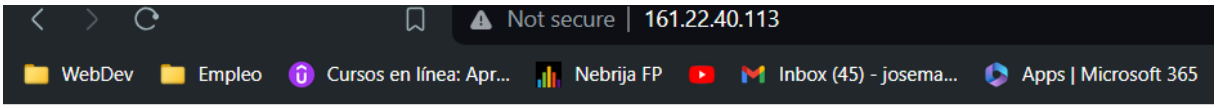
14. After this our server is running in the public ip that clouding.io gives us. The next steps we are gonna take is to actually bring our files from our local machine into the server so our project is served as well as creating our database tables. Anyway our server is now up and running.



15. Now we will bring all of our project code to our server in order to leverage our app. For this we will use PuTTy in order to copy files using pscp following these steps.
    a. I made a copy of the project in a new folder (to avoid all .git invisible files to be also copied into the server).
    b. Once I created the new folder with the folders I need I used the Windows command line to run the following pscp command: ```pscp -P 22 -r "C:\xampp\htdocs\proyecto-integrador*.*" root@161.22.40.113:/var/www/html/ ```(you can also compress everything into a .zip file and then unzip it in the server). This command will copy all the files inside the project folder and copy it to the server folder where they are needed. It is important to note that previously I removed the default

index.html file in the server folder using the command ``` rm /var/www/html/index.html ``` removing the file and making the folder empty and ready for copying my files.

c. Once all the files were copied I ran into the following issue:



Composer detected issues in your platform: Your Composer dependencies require a PHP version ">= 8.1.0".

This happens because the version that comes with the Ubuntu server is the 7. 4. To fix this what I will do is also install a newer version of PHP. Note that I will not delete the default version that comes with the server because some packages need that specific version to work, what I will do is also install the version I need for my packages to work. For this I run the following commands:

```
root@proyecto-integrador:/var/www/html# apt install software-properties-common
```

```
root@proyecto-integrador:/var/www/html# add-apt-repository ppa:ondrej/php
```

```
root@proyecto-integrador:/var/www/html# apt update
```

```
root@proyecto-integrador:/var/www/html# apt upgrade
Reading package lists... Done
Building dependency tree
Reading state information... Done
Calculating upgrade... Done
The following packages were automatically installed and are no longer required:
  libapache2-mod-php7.4 liblua5.2-0 linux-headers-5.4.0-122 linux-headers-5.4.0-122-generic linux-headers-5.4.0-137 linux-headers-5.4.0-137-
Use 'apt autoremove' to remove them.
The following NEW packages will be installed:
  debsuryorg-archive-keyring libapache2-mod-php8.3 liblua5.3-0 php8.3 php8.3-cli php8.3-common php8.3-mysql php8.3-opcache php8.3-readline
The following packages will be upgraded:
  apache2 apache2-bin apache2-data apache2-utils libapache2-mod-php libapache2-mod-php7.4 libaprl libaprutil1 libaprutil1-dbd-sqlite3 libapr
  php-common php-mysql php7.4 php7.4-cli php7.4-common php7.4-json php7.4-mysql php7.4-opcache php7.4-readline
27 upgraded, 9 newly installed, 0 to remove and 0 not upgraded.
Need to get 12.6 MB of archives.
After this operation, 23.6 MB of additional disk space will be used.
W: Sources disagree on hashes for supposely identical version '2024.02.05+ubuntu20.04.1+deb.sury.org+1' of 'debsuryorg-archive-keyring:amd64
W: Sources disagree on hashes for supposely identical version '2024.02.05+ubuntu20.04.1+deb.sury.org+1' of 'debsuryorg-archive-keyring:amd64
Do you want to continue? [Y/n] y
```

Now PHP has been upgraded to version 8.3.6.

```
root@proyecto-integrador:/var/www/html# php -v
PHP 8.3.6 (cli) (built: Apr 11 2024 20:23:19) (NTS)
Copyright (c) The PHP Group
Zend Engine v4.3.6, Copyright (c) Zend Technologies
    with Zend OPcache v8.3.6, Copyright (c), by Zend Technologies
root@proyecto-integrador:/var/www/html#
```

16. After this I installed Composer package manager for this I followed the official tutorial at https://getcomposer.org/download/
17. Once Composer was successfully installed I installed the project dependencies. For it I did the following (be careful and do not execute this commands with sudo privileges):
    a. Change directories to the project directory using ```cd /var/www/html```
    b. Run command ```composer install``` to install all needed packages.
    c. Run command ``composer upgrade`` to update possible old packages.
    d. Run command ``composer dump-autoload`` to generate new composer autoload optimized files.

```
josema@proyecto-integrador:/var/www/html$ composer dump-autoload
Generating autoload files
Generated autoload files
```

18. After this I changed the apache2.conf file to generate my desired configuration for the server, on it I permitted the .htaccess to overwrite directives in order to parse the incoming requests to my index.php. For this I did the following:
    a. Command ``sudo nano /etc/apache2/apache2.conf . This command will open with administrative privileges the config file and will let me modify it. I wrote the following Directives:

```
<Directory />
        #Options FollowSymLinks
        AllowOverride All
        Require all denied
</Directory>

<Directory /usr/share>
        AllowOverride None
        Require all denied
</Directory>

<Directory /var/www/html>
        AllowOverride All
        Require all granted
</Directory>
```

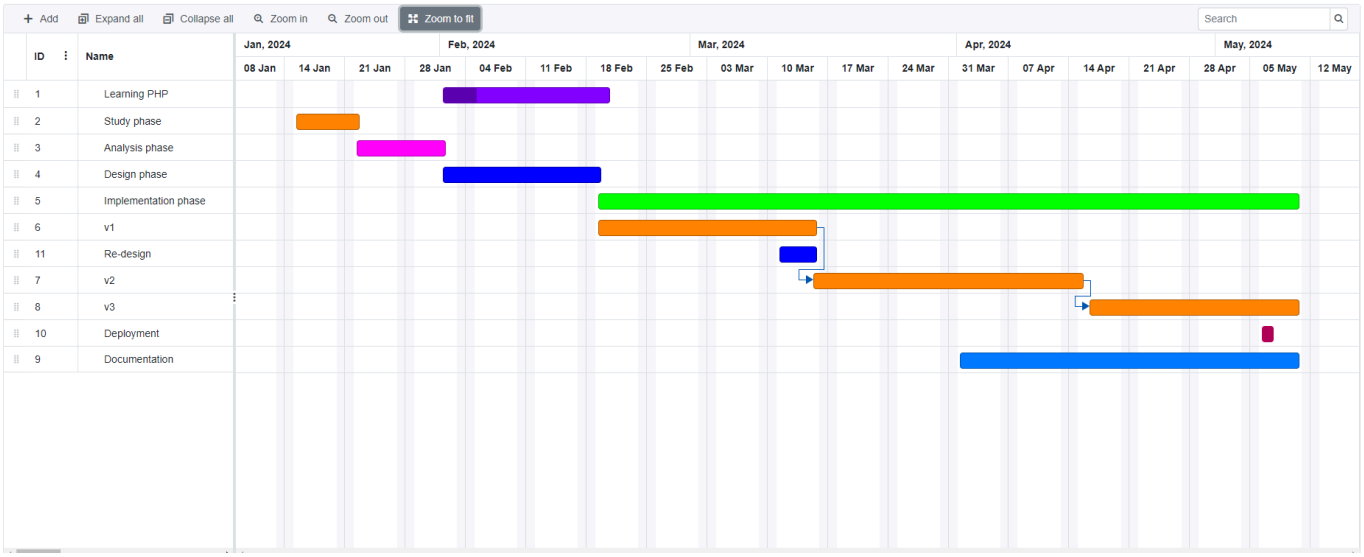
```
AccessFileName .htaccess

#
# The following lines prevent
# viewed by Web clients.
#
<FilesMatch "^\.ht">
        Require all denied
</FilesMatch>

<FilesMatch ".env">
        Require all denied
</FilesMatch>
```

19. Once I did this I realized my server was not able to parse PHP because they were not included in the LoadModules from the server. For this I enabled the PHP module for apache using command ``sudo a2enmod php`` and then running ``sudo systemctl restart apache2`` to restart the server.
20. After that I created the database, for this I did the following:
    a. Connect to mysql server with ```mysql -u root -p```
    b. Run the query for creating the database, you can find it in the docs folder of the github repository.
21. Once I did this I got another problem, my database was not connecting to the webpage. I realized that it was because the authentication method for the user was auth_socket and there was some problem with the authentication from the webpage, for solving it I did the following:
    a. For changing the mysql user authentication method into password authentication. I connected to the mysql server using the command. ```mysql -u root -p```

   b. Run the query ```ALTER USER 'root'@'localhost' IDENTIFIED WITH mysql_native_password BY 'new_password'; FLUSH PRIVILEGES;```
   c. What this query has achieved is to change the authentication method to password and set it. Now in the .env we have to change the credentials for the database in order to access it. For this I run the command ```sudo nano /var/www/html/.env```. Fill the .env variables with your data.
22. After this my webpage was able to connect to the database but I had no data on it, so I ran my seed protocol setting the .env variable ENVIRONMENT=dev and visiting the /seed route. This seeded the database with the demo data. After this I changed the ENVIRONMENT variable and this route is not accessible any more. You can read the instructions for this seed protocol in the README.md file in the Github repository.
23. The following problem that I found was that it was not possible to upload the dishes images to the storage folder in my server. I figured out that the file system the privileges were not allowing that, for this I granted access to modify the file system in the storage folder where the images will be stored. For this I ran the command ```sudo chmod -R ugo+rwx /path-to-storage```
24. After all this the server is running and fully functional. It may be that I accidentally skipped some minor step because I am documenting all this after a whole day of trying to make the server work, so it is possible that some minor step was forgotten but I think that should summarize all.

7. Gantt Diagram

In this section I will showcase how the development process has gone. For this a Gantt Diagram will be used.



1. Study phase: this was the first phase of the project, although here it appears taking like 1 week it actually took longer, but for display reasons I can not showcase how long it took but approximately it took one month to think about the idea, plan the rest of the steps to take, etc.
2. Analysis phase: in this phase I gathered the requirements of the app and adjusted it to the project size and purpose. I defined the functionalities as well as decided which technologies to use and more or less how much time each phase would take. It took approximately 9 days.
3. Learning PHP: after the analysis phase I decided that the project would be made using PHP, and, as I had no prior experience with PHP I started learning it. It took approximately 2 weeks of course[1]. After these 2 weeks I started to feel confident and did what is the best to learn a programming language, build something with it, so I started programming the prototype of the project. I did not want to get stuck in a course hell so I started programming as soon as I had some knowledge about the language and learned through the programming path.
4. Design phase: concurrently with learning PHP I started designing the architecture, structure of the project, database, classes, etc. It took approximately 13 days to fully design the webpage. Notice that the visual design of the webpage is inspired by one of my side projects, so I did not have to design a lot in these regards which sped up the design phase.

5. Implementation phase: took a total of 60 days and 3 iterations to get the finished product.
    5.1. v1: this phase took 20 days. In it I started programming the core of the app, applied what I started learning about PHP and created the MVC (Minimum Viable Product). This phase was a pretty early development phase and its purpose was to leverage the core structure of the project as well as spot some design problems.
    5.2. Re-design: I realized that the database design had some problems, as explained in the database design section I changed the design from a pure relational model to a mixed model with some NoSQL features. This redesign also came with some structural changes, mainly in the folders structure. I also spotted that during some tests some sensitive information was committed to the GitHub repository while testing the image upload functionalities (some personal documents where uploaded and committed), so I archived the repo, made it private and started a new one with the code from the older one.
    5.3. v2: the code from this version and upwards is the one that you have access to in the GitHub repository. In this version the database structure was updated and all the functionalities were implemented. This phase took 22 days.
    5.4. v3: this is the final version, in it I adapted the web design to be responsive in the key pages, index, menu, cart and log in. I refactored some code and created some important classes like the AuthMiddleware, AuthSession, ServerErrorLog and ResponseStatus and I added some testing to the app. This is the version that has been deployed.
6. Deployment: it took 1 day, I created a VPS in clouding.io and installed all necessary software via SSH PuTTy commands.
7. Documentation: It has taken approximately 30 days to come up with this full documentation, I have been working on it alongside when I was programming the project.

8. Next Steps

The project has come to an end but this does not mean it can not be improved. This project has been designed to match the Nebrija Institute requirements and deadlines. In doing so there are a lot of functionalities and features that were not introduced in the project because it would come out of the scope of this project. I will like to showcase which steps I would take if I had to continue building it and improving it:

- Improving error handling in all layers.
- Refactor code to include all public files inside one single folder.
- Build responsive design for the whole app and not only the main pages.
- Build the whole testing of the application making use of functional tests as well as other performance indicator tests.
- Integrate a payment platform in order to make available payments online.

9. Conclusions

It is no surprise that making a project of this size and depth really changes you. After all, I accomplished all the requirements and I am pretty happy with the results. The webpage is up and running and it is completely functional, Chef Manuelle would be able to open to the online world and sell its services for new potential clients.

Building a project like this and having to think about all the architectural aspects as well all the design aspects really help you get a better understanding of how to solve the different new problems.

During these months of preparation and execution I have encountered a lot of problems and push backs, but it is from them that we learn. I can hardly describe how facing real problems and trying to build the most accurate solutions really shift the way you view the world and solve problems and what is more important, you learn that you are capable of solving any problem ahead of you.

I have achieved all my goals for the project and I think that for being a first course project, its quality is highly remarkable. Trying to improve every day, being open to learning new programming languages and paradigms really influences the way you program in all aspects and it is one of the things I like the most about programming, you never stop learning.

10. References

1. Gio, P. W. (2020). Full PHP 8 tutorial - Learn PHP the right way in 2024 [Video]. In YouTube. https://www.youtube.com/watch?v=sVbEyFZKgqk&list=PLr3d3QYzkw2xabQRUpcZ_IBk9W50M9pe-&index=1

2. Perera, I. (2023, June 7). Introduction to object-oriented programming: Explaining the core principles and concepts of OOP. *Medium*. https://medium.com/@ishankaperera20/introduction-to-object-oriented-programming-explaining-the-core-principles-and-concepts-of-oop-95ee24050a2e

3. Ingalls, S. (2021, November 17). What is a client-server model? A guide to client-server architecture. *ServerWatch*. https://www.serverwatch.com/guides/client-server-model/

4. GfG. (2020, June 29). HTTP full form. *GeeksforGeeks*. https://www.geeksforgeeks.org/http-full-form/

5. Hernandez, R. D. (2021, April 19). The model view controller pattern – MVC architecture and frameworks explained. freeCodeCamp.Org. https://www.freecodecamp.org/news/the-model-view-controller-pattern-mvc-architecture-and-frameworks-explained

6. *HTML5 - MDN Web Docs Glossary: Definitions of Web-related terms*. (n.d.). MDN Web Docs. Retrieved April 30, 2024, from https://developer.mozilla.org/en-US/docs/Glossary/HTML5

11. Contact Data

- Email: jm3develop@gmail.com
- Portfolio: https://www.jm3.dev
- Linkdin: https://www.linkedin.com/in/josemanuelmontanomengual/