# GROUP PROJECT FRONT SHEET

| | | | |
|---|---|---|---|
| **Qualification** | **BTEC Level 5 HND Diploma in Computing** | | |
| **Unit number and title** | WEBG301 - Project Web | | |
| **Submission date** | 1 – July – 2022 | **Date Received 1st submission** | |
| **Re-submission Date** | | **Date Received 2nd submission** | |
| **Student Name** | Trinh Duc Anh | **Student ID** | GCH210829 |
| **Class** | GCH1002 | **Assessor name** | Nguyen Dinh Tran Long |

**Student declaration**

I certify that the assignment submission is entirely my own work and I fully understand the consequences of plagiarism. I understand that making a false declaration is a form of malpractice.

| | |
|---|---|
| **Student's signature** | |

□

**Grade**

□ Summative Feedback: □ Resubmission Feedback:

Grade:

Assessor Signature:

Date:

Signature & Date:

# Contents

# I. User requirements

## 1. User Stories

| No | User story | Functional/ Nonfunctional | Explanation |
|---|---|---|---|
| 1 | As a customer, I want to be able to create an account so I can put products into a cart and checkout with them | Functional | This is important because the customer might want to pick up products at one time and checkout at another. As developers of this project, this feature should absolutely implemented to accommodate for the good experience of the customer. |
| 2 | As a customer, I want to be directed to the login page if I happen to click on the "Cart" button when I am not logged in | Functional | Each cart is attached to a different customer profile. So in order to add an item into a cart, the user needs to login with a customer account |
| 3 | As a customer, when I click on the "Checkout" button, I want a receipt to be generated based on the items I have added into my cart and the information on my account | Functional | This feature is what brings the revenue to the online store, making it an essential part of the website. |
| 4 | As a customer, when I hover over the picture of a product, I want brief information about the product to be displayed | Functional | In the product index page, the list of products would only display images of the products, implementing this feature would let the visitor know more about the product without cluttering the page with pieces of text, ruining the aesthetics of the website |
| 5 | As a customer, when I click on a product, I want to be taken to a detail page that presents all of the product's information along with a "Add to cart" button | Functional | The brief information on the product list page is not enough to give the visitor all the information about the product along with all of its pictures. A detail page is needed to give the visitor more details about the product. After reading and considering the product, the customer can add the product to their cart. |

|  |  |  |  |
|---|---|---|---|
| 6 | As the operation manager, I want the website to have aesthetics that suit the store's theme, which is artistic hand-made crafts | Non-Functional | A website with aesthetics that matches the store's products and is unique is an important aspect of a website as it improves the shopping experience for the customer and from there increase sales. |
| 7 | As a system admin, I want additional links to appear on the navigation bar when an admin is logged into the website | Functional | This feature would help the admins navigate through the admin functions from the homepage. |
| 8 | As a system admin, I want to be able to perform CRUD on different information of the database without having to directly interact with the database | Functional | This feature allows admins who are not able to program to modify the data on the system without having to work directly with the database as doing so would require programming skills and could lead to human error. Furthermore, implementing CRUD sites with UI gives the admin a more intuitive user experience in performing the task |
| 9 | As the operation manager, I want the system to encrypt the login information when a new user is created in order to protect the information of everyone | Non-Functional | This feature is a step in cyber security. Should a hacker have breached through the firewall and hacked into the system, the encrypted data would be able to protect it from being brute forced for up to hundreds of years |
| 10 | As a customer, I want my account to be automatically logged in after I registered for a new account | Functional | This feature improves the user interaction of the user with the system. Instead of having to navigate to the login page and enter the account information again to log in, wasting the customer time. |

## 2. Use-case diagram

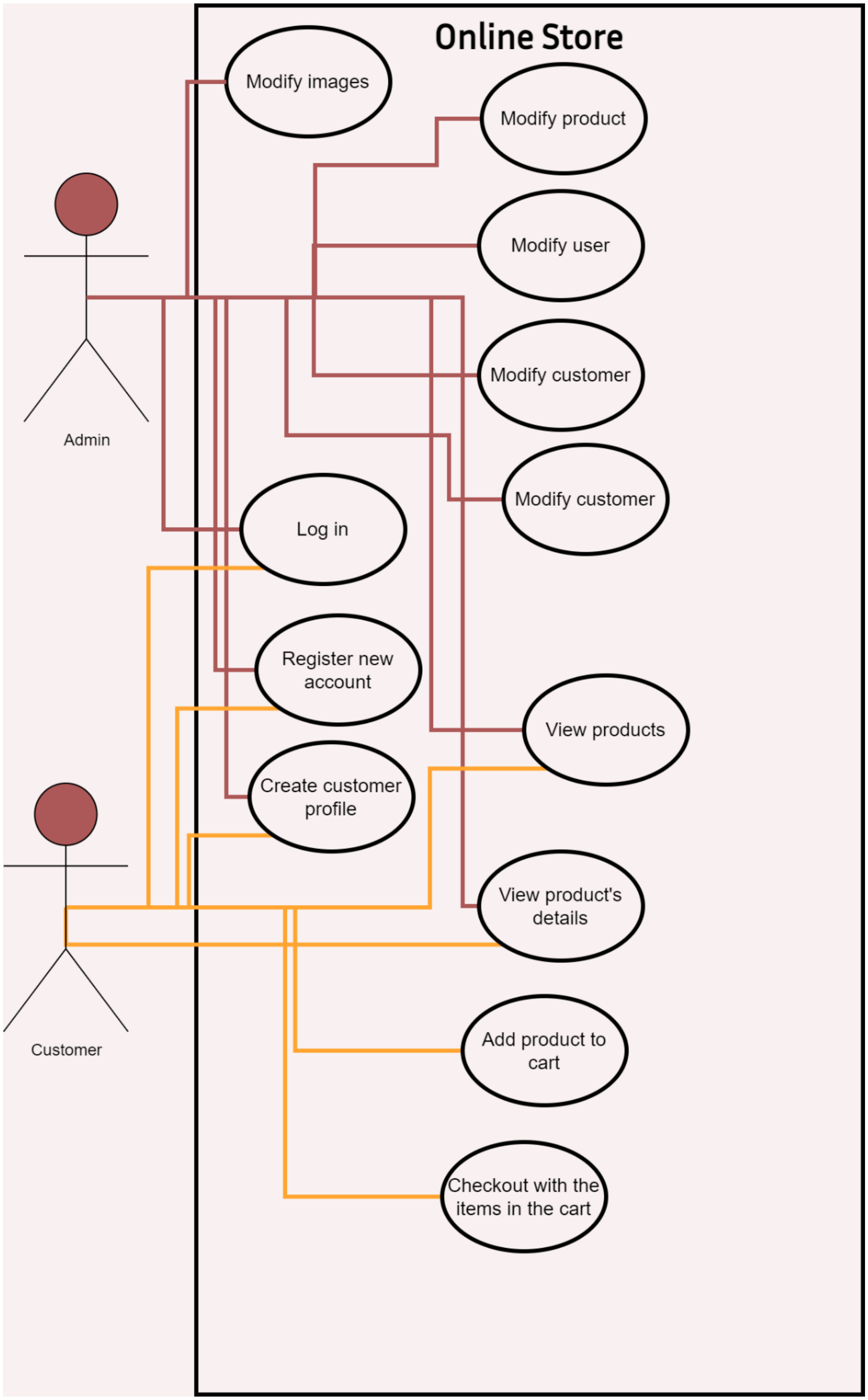


*Figure 1 Use-case diagram*

# II. System Design

## 1. Site map

The project sitemap starts from the home page (Home), which is the main website for users to access and manipulate functionality. On the home page, users will branch out to other functional websites including: Login, Product, Cart, and the admin-specific function page, which is the Update by admin page.

- Login page: This is the page that helps users to login. There will be two types of decentralized accounts for User and Admin. In login there will be login, logout and register branches for newly registered users.
- Product page: contains information, general images of all products of the website. This page will guide users to access the Product detail which contains detailed information of each product. There will also be a Category so that users can filter products by the corresponding category.
- Cart page: When the user chooses to add to the cart, the product will be included in the Cart detail where the product information the user chooses to buy. Inside there will be a Checkout page to output the invoice of the order.
- Update by Admin page: special page for Admin users. The page will branch into many different functional pages that have crud functions for each entity of the project such as: product, user, employee, etc.

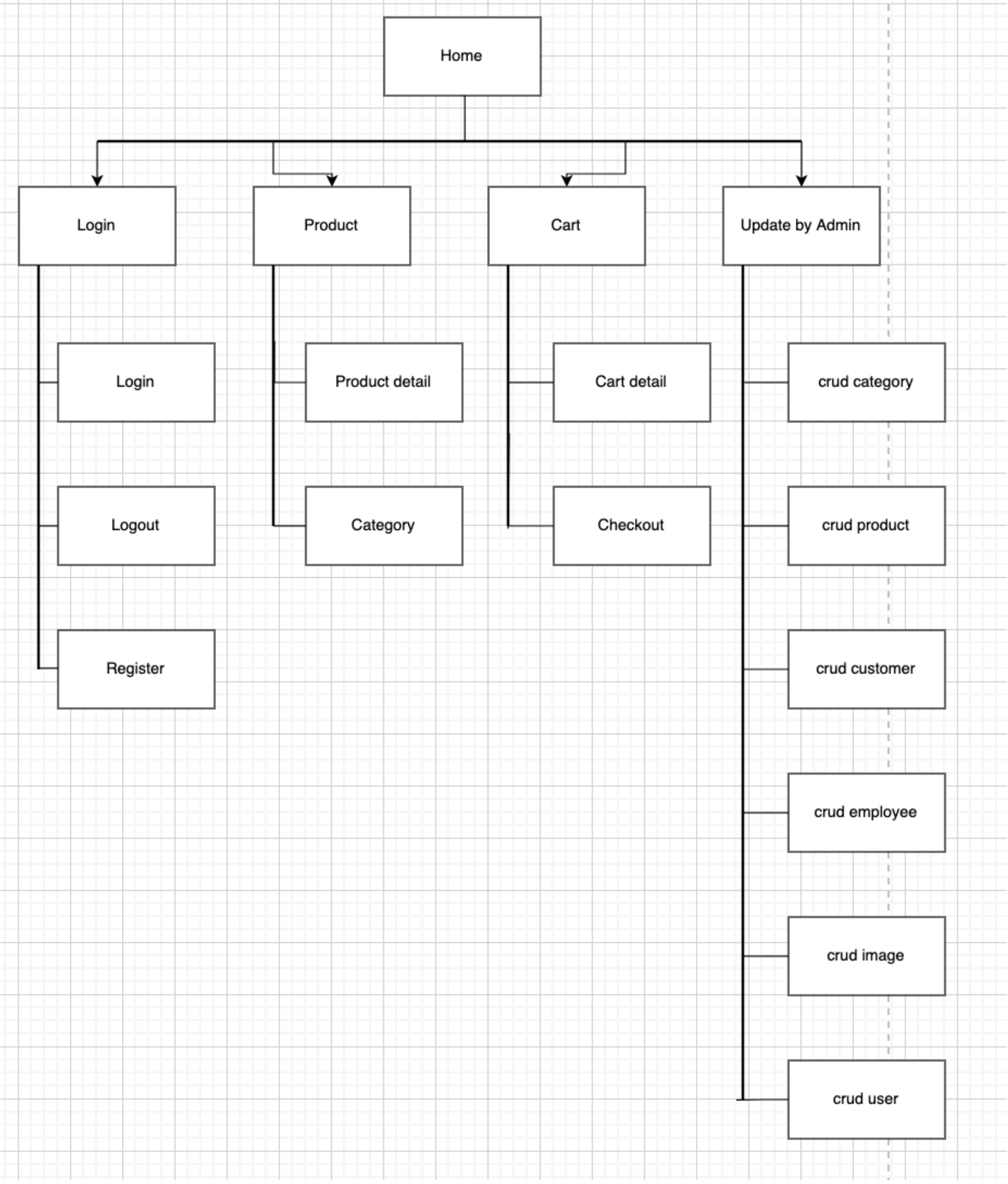


*Figure 2 Sitemap*

## 2. Entity Relationship Diagram

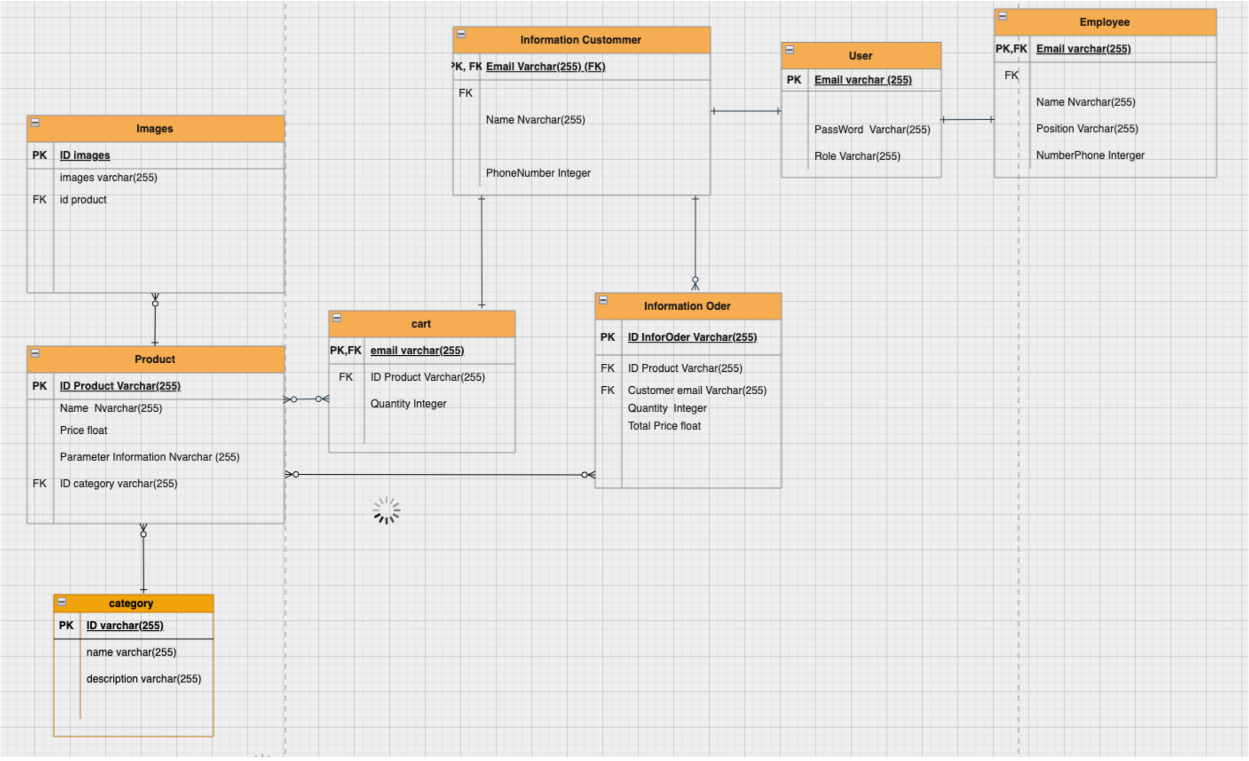


*Figure 3 ERD*

Relationship:

| Relation 1 | Relation 2 | Type of relationship | Reason |
|---|---|---|---|
| **User** | Employee | One to One | One user account for one employee and one employee has only one user account |
| **User** | Customer | One to One | One user account for one customer and one customer has only one user account |
| **Customer** | Order Infomation | One to many | A customer has many different Orders and an Order of only one Customer |
| **Customer** | Cart | One to One | One customer for one cart and one cart created by only one Customer |
| **Product** | Cart | Many to many | One product is in multiple carts and one cart can contain many products |
| **Product** | Order Infomation | Many to many | One product is in multiple Order Infomation and one Order Infomation can contain many products. |
| **Product** | Category | Many to One | A Product has only one category, but a category can contain many products |
| **Product** | Image | One to Many | A product has many photos, but one photo is only of that product |

*Table 1 Relationship between entities*

# 3. Wireframes

## 3.1. Homepage

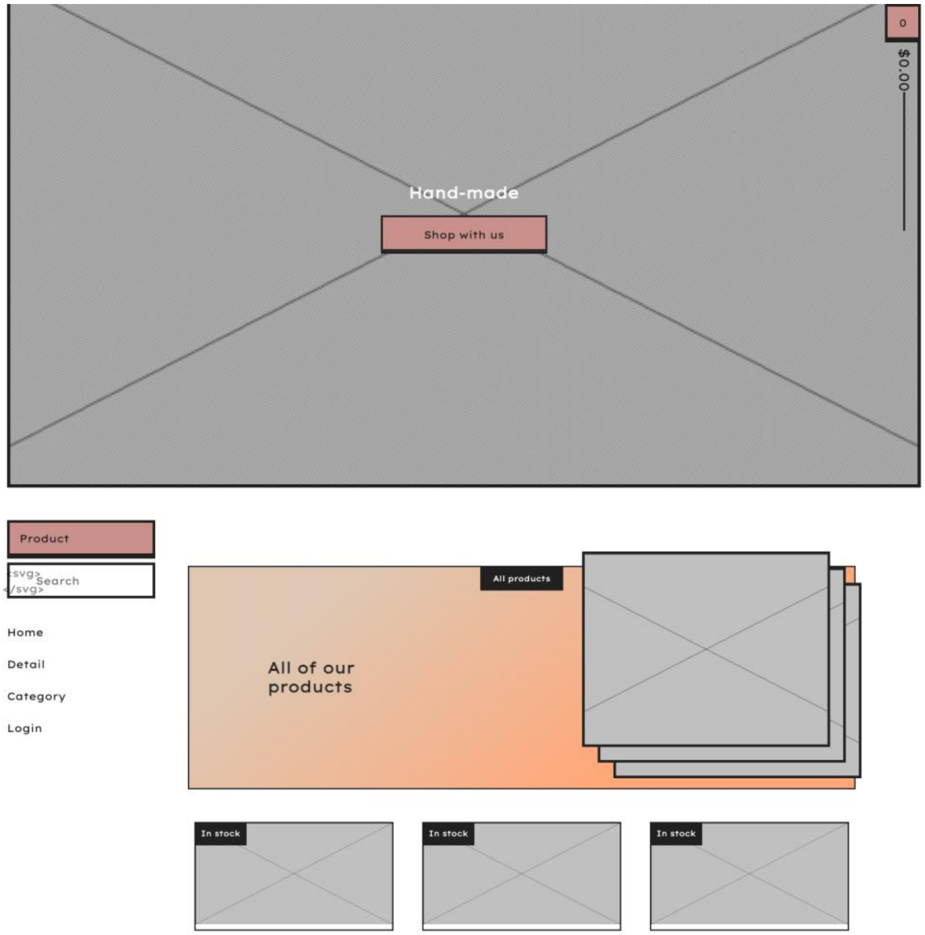
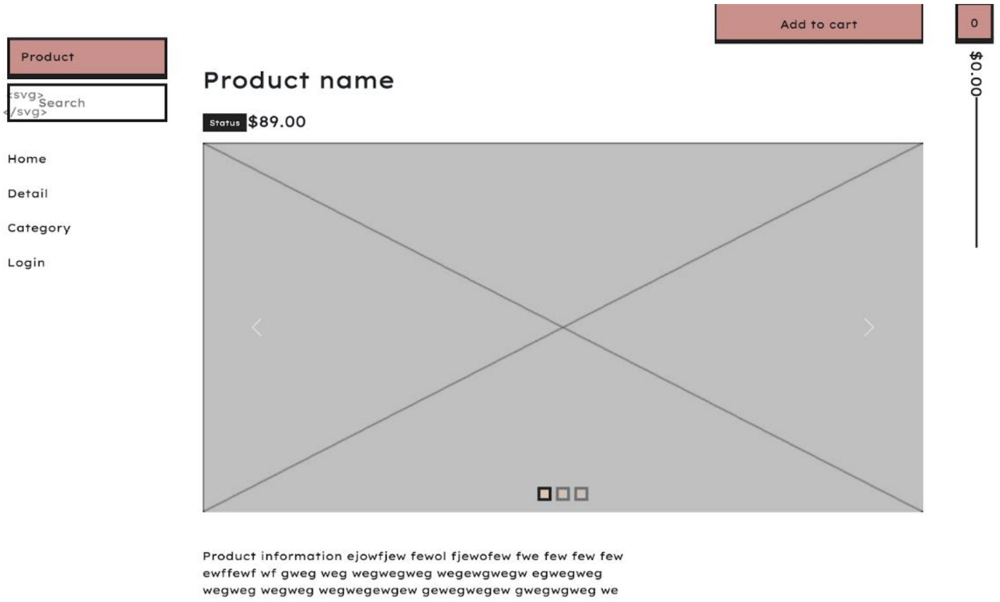


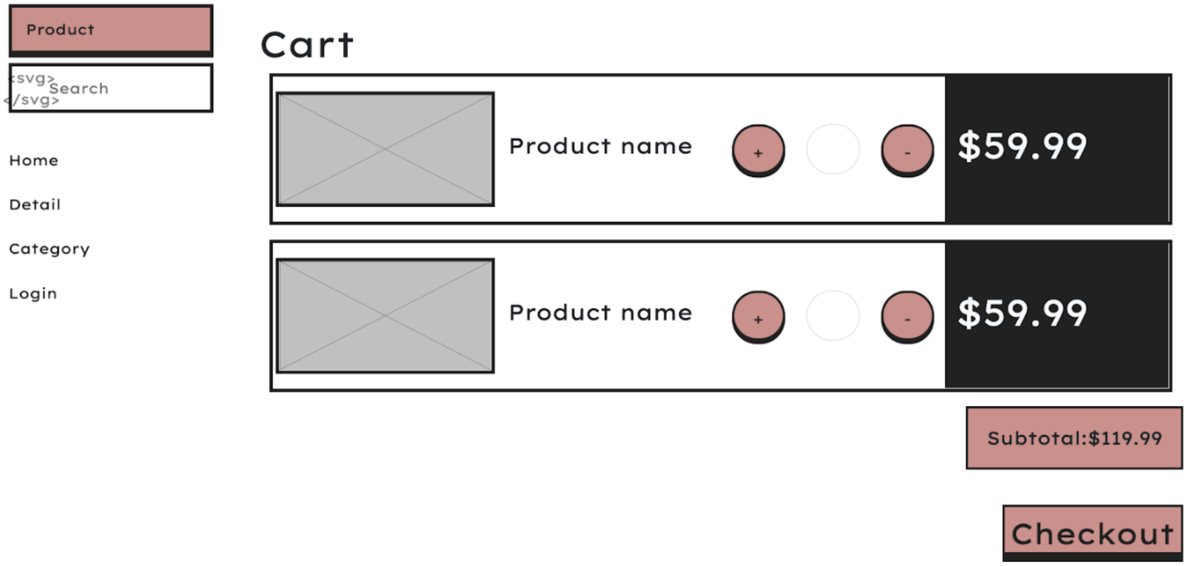
*Figure 4 Homepage*

## 3.2. Product detail

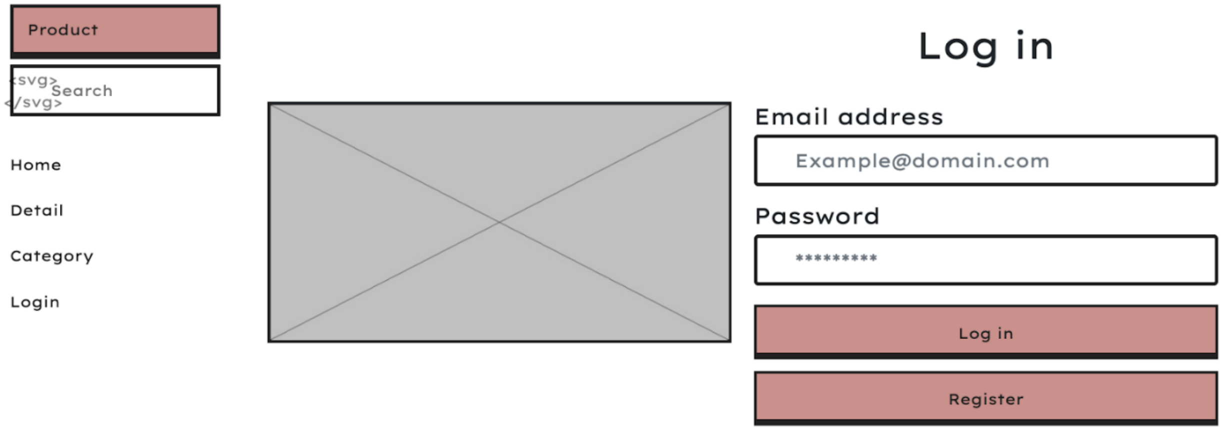


Product information ejowfjew fewol fjewofew fwe few few few ewffewf wf gweg weg wegwegweg wegewgwegw egwegweg wegweg wegweg wegwegewgew gewegwegew gwegwgweg we

## 3.3. Cart



### 3.4. Log in



## Log in

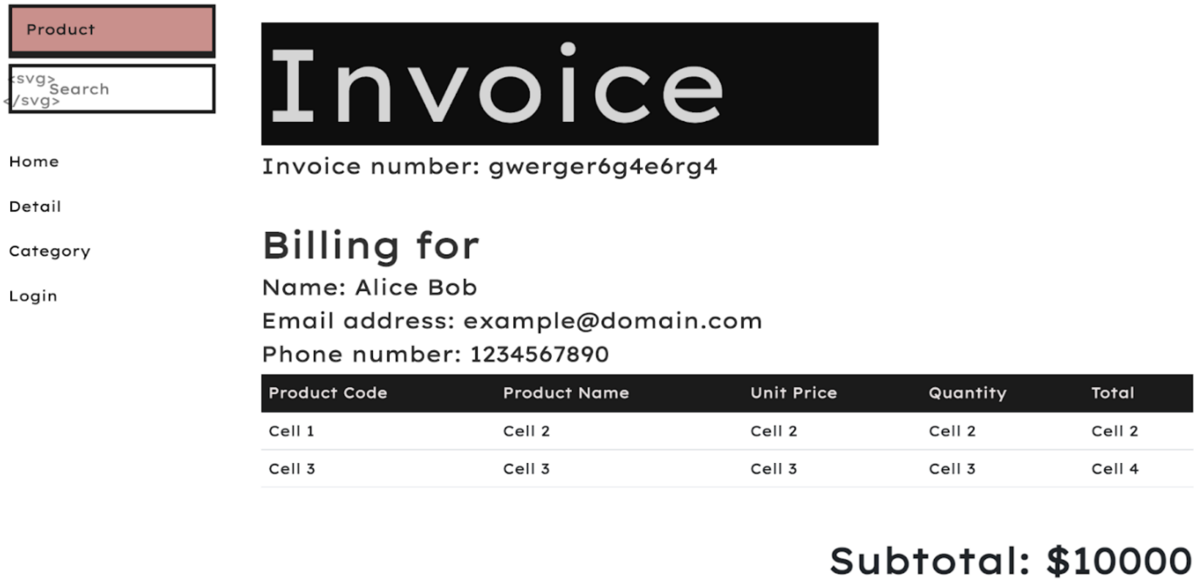
Email address

Example@domain.com

Password

*********

### 3.5. Checkout



# Invoice

Invoice number: gwerger6g4e6rg4

## Billing for

Name: Alice Bob

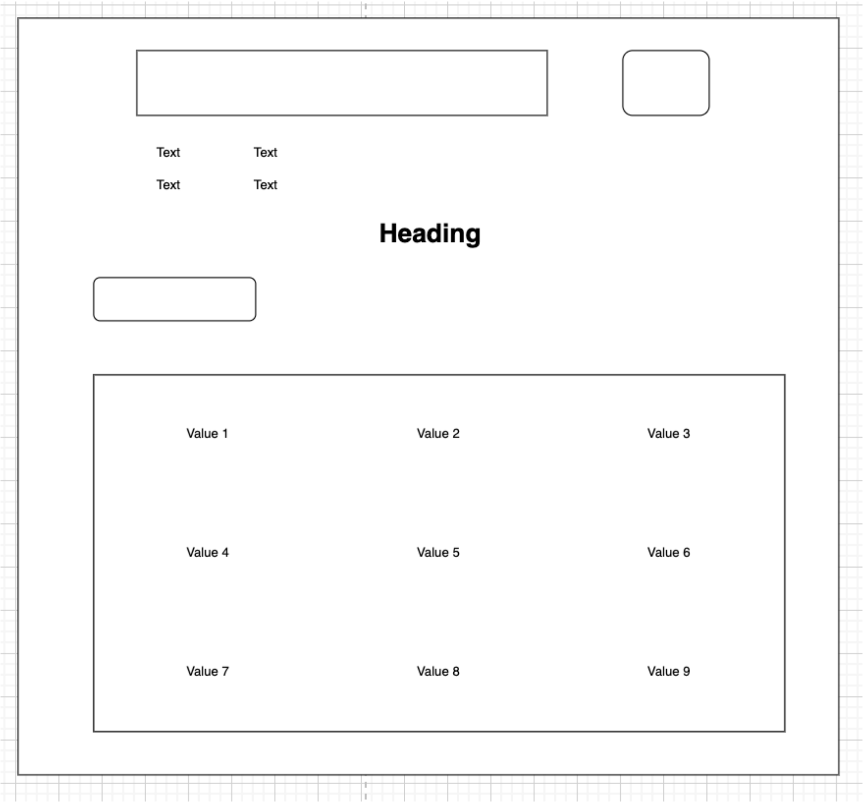
Email address: example@domain.com

Phone number: 1234567890

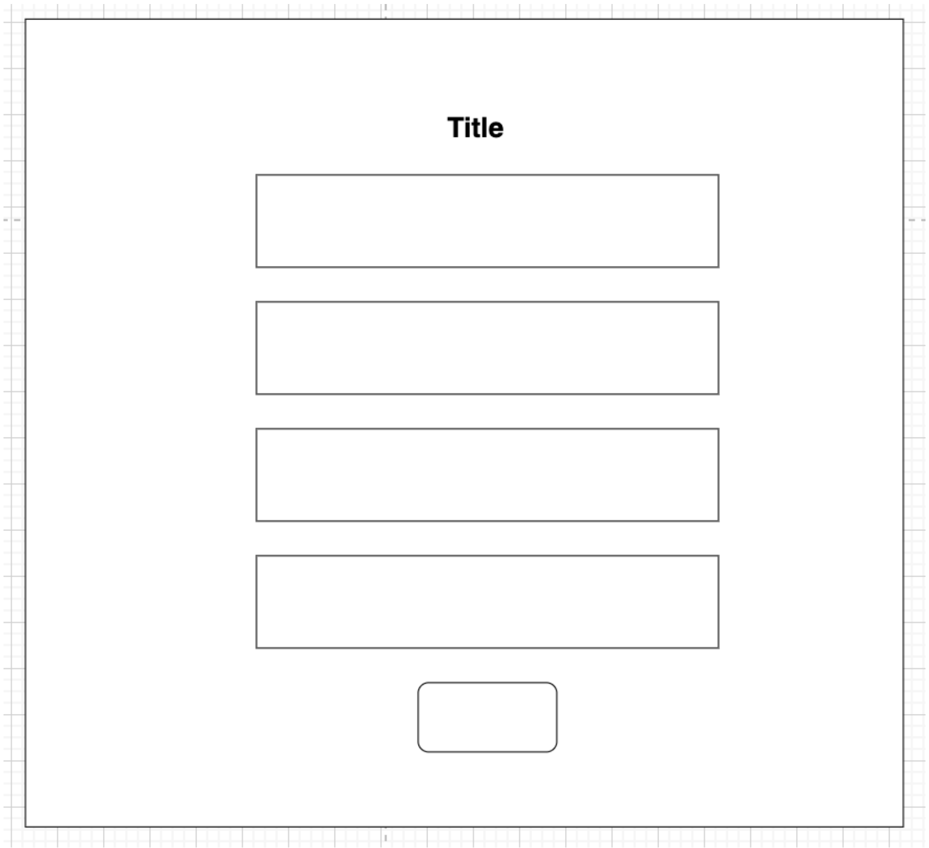
| Product Code | Product Name | Unit Price | Quantity | Total |
|---|---|---|---|---|
| Cell 1 | Cell 2 | Cell 2 | Cell 2 | Cell 2 |
| Cell 3 | Cell 3 | Cell 3 | Cell 3 | Cell 4 |

## Subtotal: $10000

### 3.6. Product list



### 3.7. Product add, edit



# III. Implementation

## 1. Source code

### 1.1. MVC design pattern

Model-View-Controller (MVC) design is a practice where the developers split up the code into multiple files. Each file handles a number of certain tasks.

Controller is what the client would be directly interacting with. It determines what data, what file is delivered to the client, and what input to take from them. It modifies the View component to display appropriate information to the client and creates/modifies data based on the logical design provided by the Model.

View is the User Interface design for the user/client to look at and interact with visually. The View file takes data from the Controller to display to the user and takes the data from the user and passes it to the Controller

Model is the component where the structure of the data is defined. It also contains methods that allow access to modify or view its data so that it can be called by either the View or the Controller.

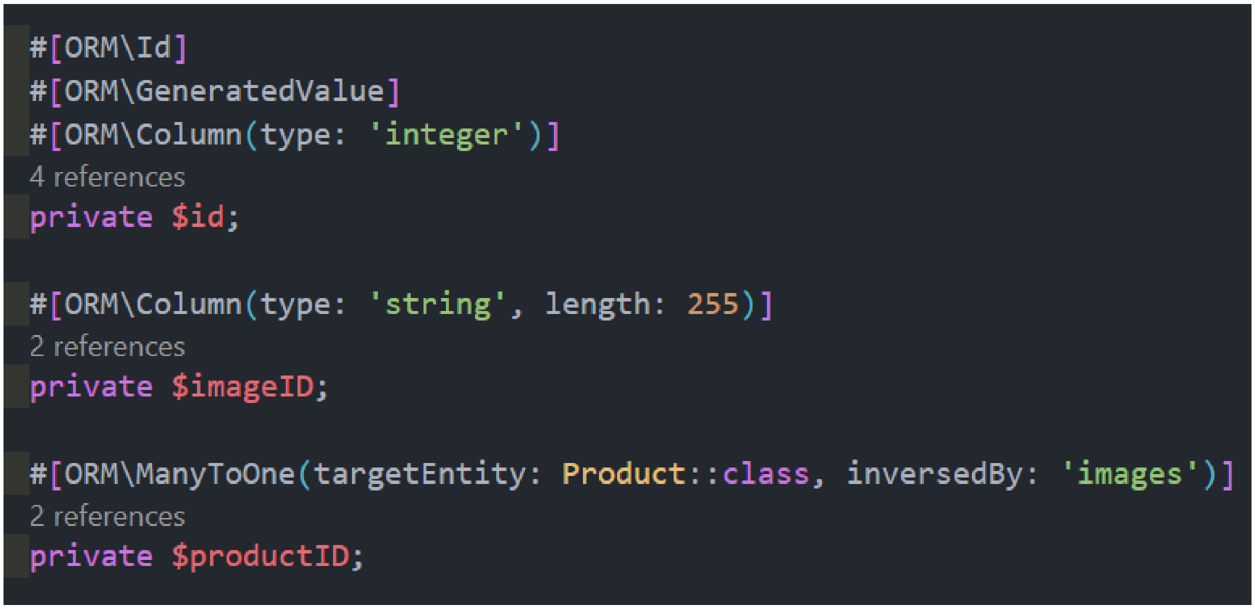
## 1.2. MVC in the project

### *1.2.1.Model*

In this project, the Model is used to define the data structure of these entities along with methods to manipulate their data:

- Cart
- Category
- Customer
- Employee
- Image
- OrderInfo
- Product
- User

For example, the entity Image consists of three variables, they are id, imageID, and productID.



```
#[ORM\Id]
#[ORM\GeneratedValue]
#[ORM\Column(type: 'integer')]
4 references
private $id;

#[ORM\Column(type: 'string', length: 255)]
2 references
private $imageID;

#[ORM\ManyToOne(targetEntity: Product::class, inversedBy: 'images')]
2 references
private $productID;
```

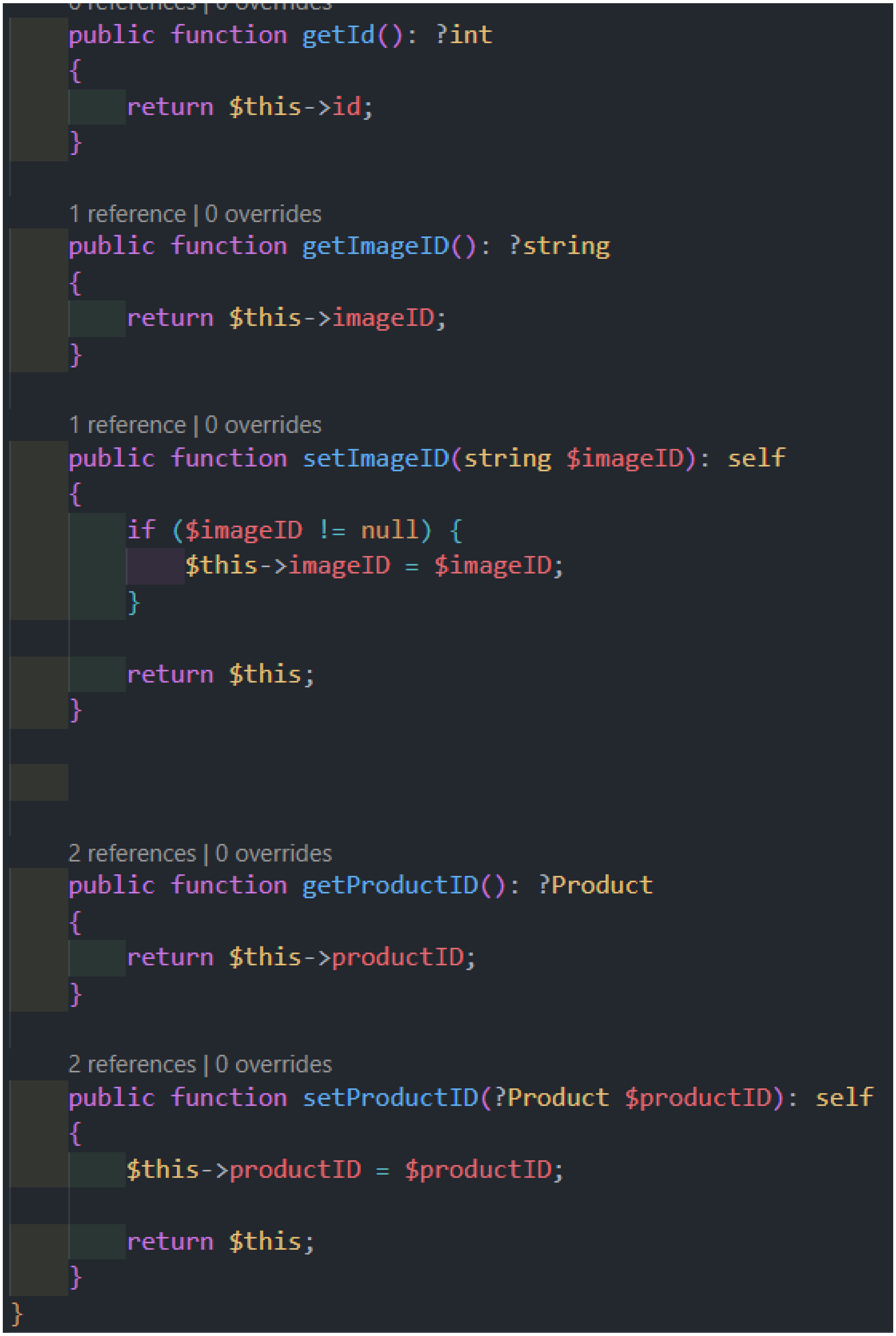
*Figure 5 Image Entity*

The variable “id” is an identifier automatically generated by Symfony when the Entity is created, its data type integer and is automatically incremented when a new instance is added into its table in the database.

“imageID” is a user-defined variable that would be used as the name for the image files that are added to the system, hence the data type of string with a maximum length of 255.

“productid” is the variable that establishes the relationship between the entity “Image” and entity “Product”. The relation between these two is Many To One, meaning that every image is associated with only one product while one product can be associated with multiple images.

After the variables, there are the getters and setter methods used to return and modify the data of an instance of this entity.

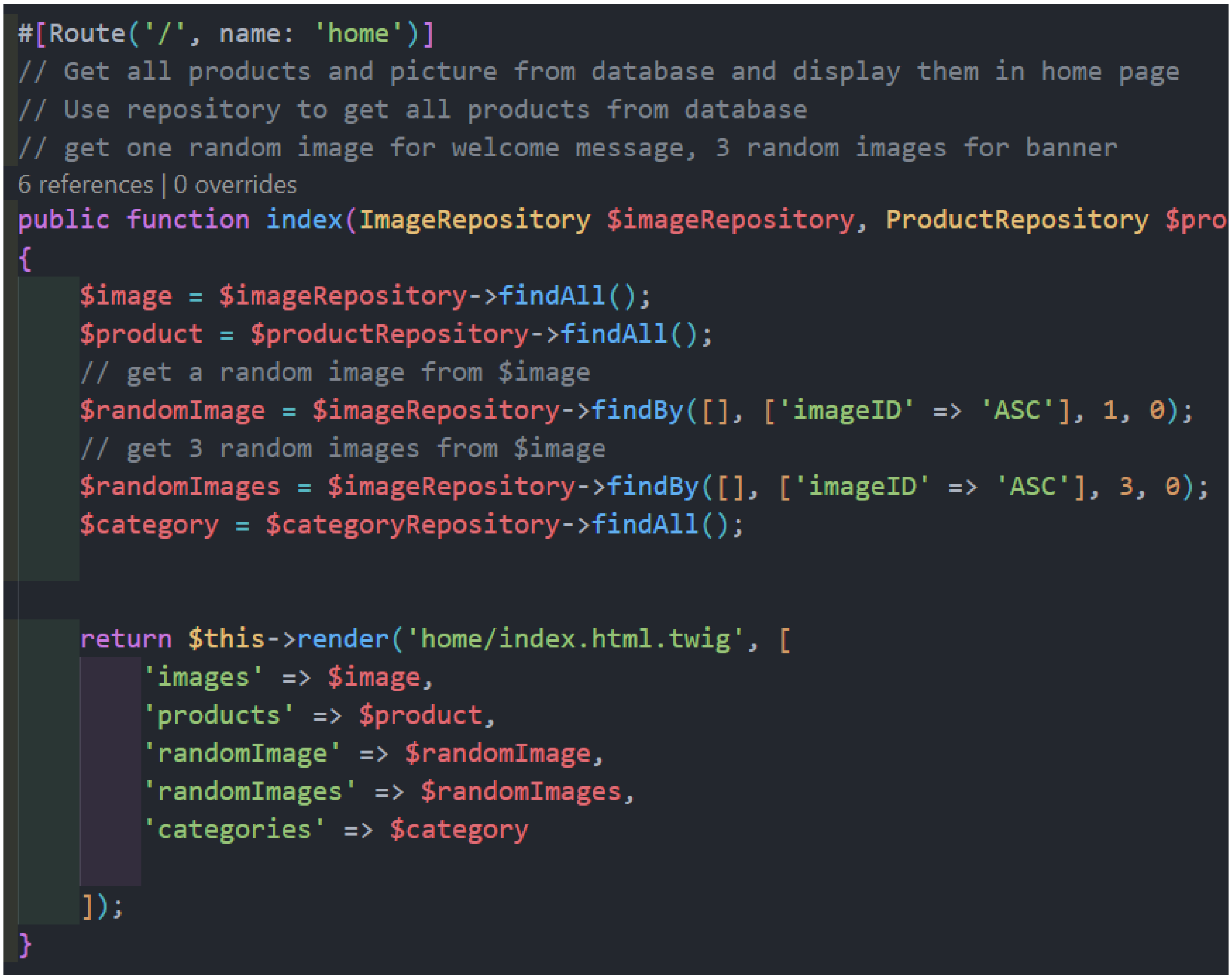


```
    0 references | 0 overrides
    public function getId(): ?int
    {
        return $this->id;
    }

    1 reference | 0 overrides
    public function getImageID(): ?string
    {
        return $this->imageID;
    }

    1 reference | 0 overrides
    public function setImageID(string $imageID): self
    {
        if ($imageID != null) {
            $this->imageID = $imageID;
        }

        return $this;
    }



    2 references | 0 overrides
    public function getProductID(): ?Product
    {
        return $this->productID;
    }

    2 references | 0 overrides
    public function setProductID(?Product $productID): self
    {
        $this->productID = $productID;

        return $this;
    }
}
```

*Figure 6 Getters and Setters for Image entity*

### 1.2.2. Controller

There are two main components of a Controller, they are routes and functions. Routes are the trigger that defines the conditions in which certain functions would be executed. For example:



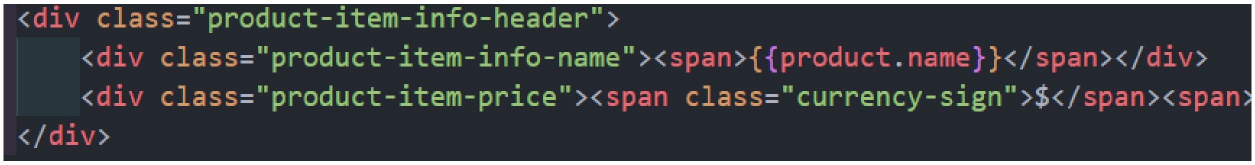
```
#[Route('/', name: 'home')]
// Get all products and picture from database and display them in home page
// Use repository to get all products from database
// get one random image for welcome message, 3 random images for banner
6 references | 0 overrides
public function index(ImageRepository $imageRepository, ProductRepository $pro
{
    $image = $imageRepository->findAll();
    $product = $productRepository->findAll();
    // get a random image from $image
    $randomImage = $imageRepository->findBy([], ['imageID' => 'ASC'], 1, 0);
    // get 3 random images from $image
    $randomImages = $imageRepository->findBy([], ['imageID' => 'ASC'], 3, 0);
    $category = $categoryRepository->findAll();


    return $this->render('home/index.html.twig', [
        'images' => $image,
        'products' => $product,
        'randomImage' => $randomImage,
        'randomImages' => $randomImages,
        'categories' => $category

    ]);
}
```

*Figure 7 Homepage route*

When the user enters the domain of the store without any extension, the function “index” is executed. In this function, the controller takes the data of every image, product, and category inside the database and assigns it to appropriate variables. This is a homepage that displays all of the products along with their images, making it necessary to gather this information from the database. Then, the function renders the file ‘home/index.html.twig’ which is a View component that displays the homepage to the user. The controller passes the variables that were assigned before to this View so it can use the data to display correct information to the user.

### 1.2.3. View

In this project, the View is written in Twig, a language that utilizes HTML in a way that can be used with the Symfony framework. For example:



```
<div class="product-item-info-header">
    <div class="product-item-info-name"><span>{{product.name}}</span></div>
    <div class="product-item-price"><span class="currency-sign">$</span><span
</div>
```

*Figure 8 Snippet code of View*

The Twig file uses regular HTML but the Twig components and functions are put into a {{}}. In this instance {{product.name}} is used to display the name of a product onto the page for the user to see.

## 1.3. Website design

To design the front-end of the website, a design language called “Neu Brutalism” is applied. Neu Brutalism is a design language that is trending in recent years. One of the most notable characteristics of Neu Brutalism is its contrast using solid borders and bold text, making everything pop.

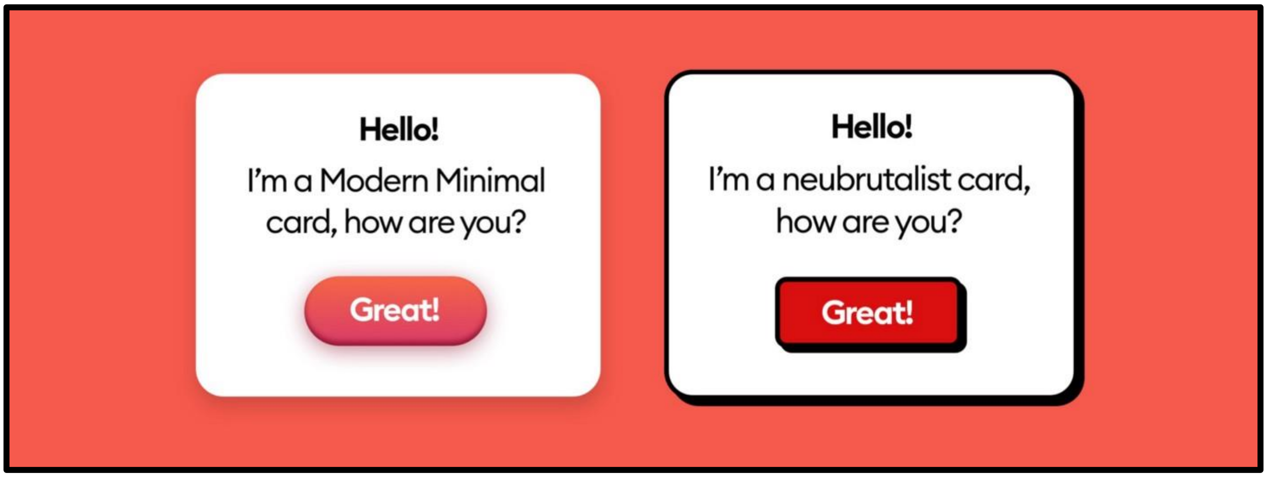

Hello!
I’m a Modern Minimal card, how are you?
Great!
Hello!
I’m a neubrutalist card, how are you?
Great!

*Figure 9 Modern Minimal and Neubrutialist design (Malewicz, 2022)*

To come up with the final design decision for the website, many online stores have been studied as a reference and some of their components are recreated using Bootstrap 5, a CSS and JavaScript library free for everyone. The websites studied are [https://negativecollection.bigcartel.com/](https://negativecollection.bigcartel.com/), and [https://paulinamocna.bigcartel.com/](https://paulinamocna.bigcartel.com/).

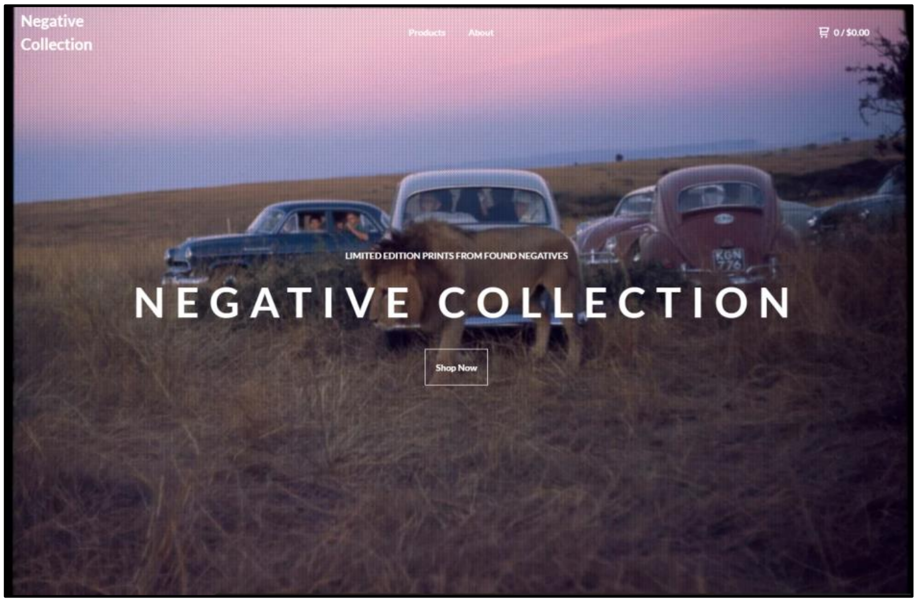
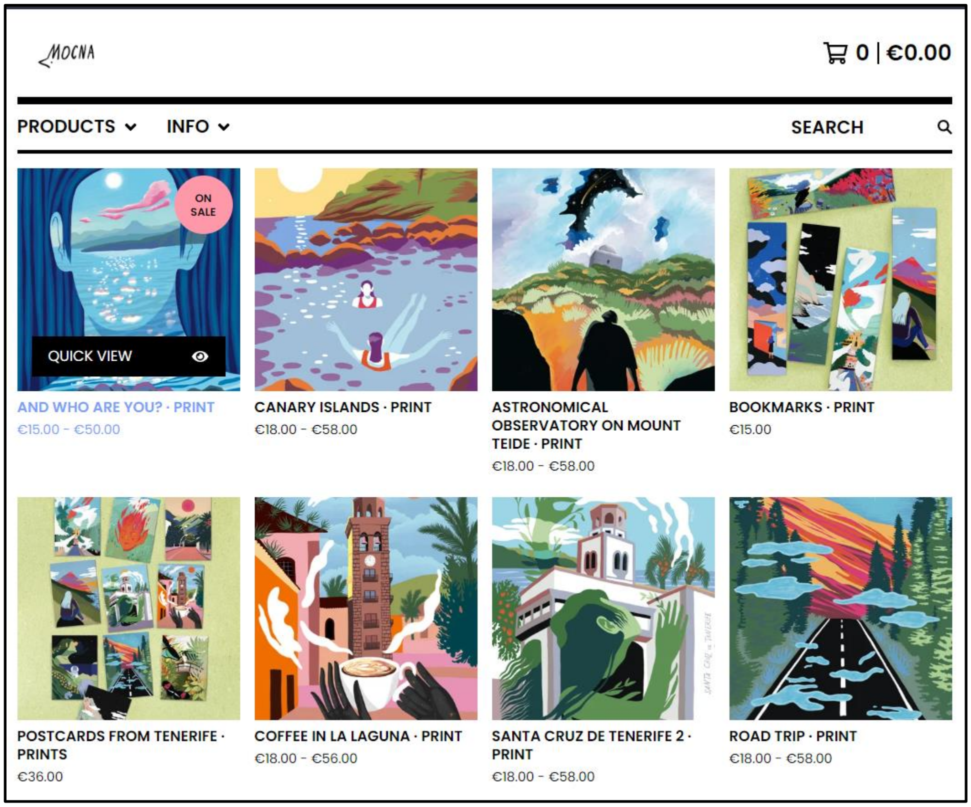


*Figure 10 Negative Collection*



AND WHO ARE YOU? · PRINT
€15.00 – €50.00

CANARY ISLANDS · PRINT
€18.00 – €58.00

ASTRONOMICAL OBSERVATORY ON MOUNT TEIDE · PRINT
€18.00 – €58.00

BOOKMARKS · PRINT
€15.00

POSTCARDS FROM TENERIFE · PRINTS
€36.00

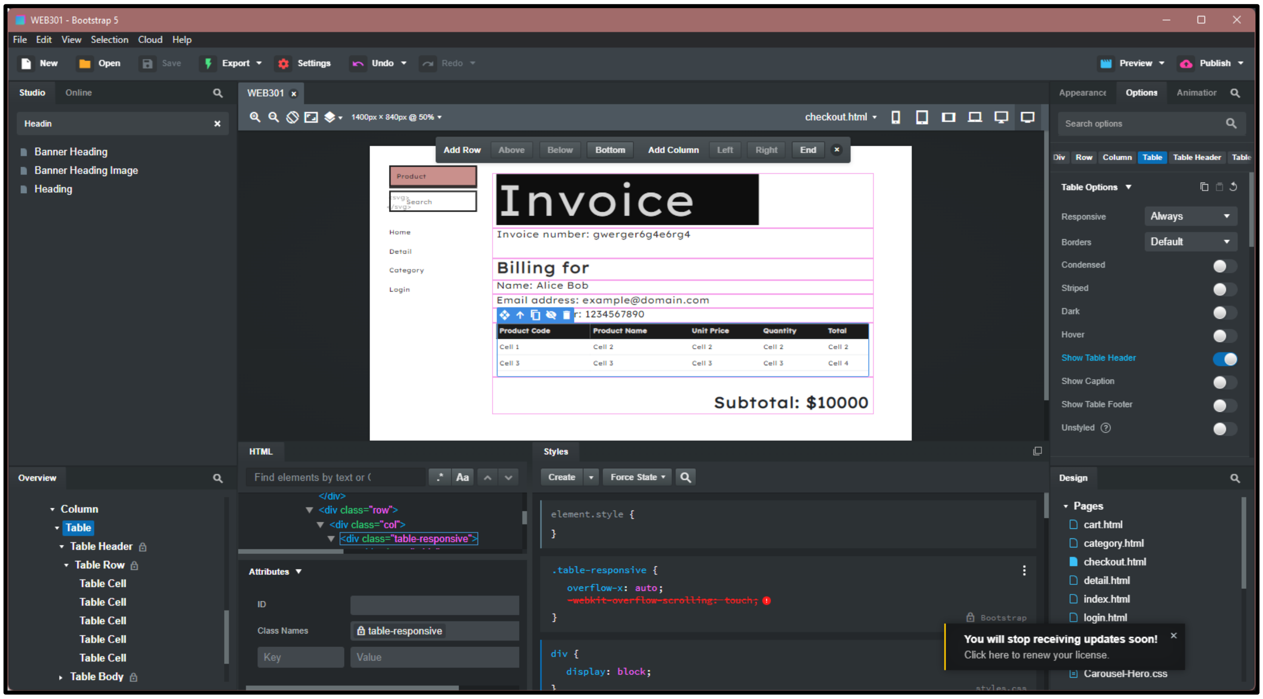
COFFEE IN LA LAGUNA · PRINT
€18.00 – €56.00

SANTA CRUZ DE TENERIFE 2 · PRINT
€18.00 – €58.00

ROAD TRIP · PRINT
€18.00 – €58.00

*Figure 11 Paulina Mocna*

These two websites were chosen because they followed the Neubrutalist design, which is the goal for this project. After they were studied, an html interface is created using an application called Bootstrap Studio 6. Bootstrap Studio is a tool that visualizes the process of creating a website design in order to help the developer create a layout more easily and mitigate human error while trying to design a website.


WEB301 - Bootstrap 5
File Edit View Selection Cloud Help
New Open Save Export Settings Undo Redo
Preview Publish
Studio Online
WEB301
1400px × 840px @ 50%
checkout.html
Add Row Above Below Bottom Add Column Left Right End
Appearance Options Animation
Search options
Div Row Column Table Table Header Table
HTML
Find elements by text or (
Styles
Create Force State
Overview
Design

Headin

- Banner Heading
- Banner Heading Image
- Heading

Product

Search

Home

Detail

Category

Login

# Invoice

Invoice number: gwerger6g4e6rg4

Billing for

Name: Alice Bob

Email address: example@domain.com

r: 1234567890

| Product Code | Product Name | Unit Price | Quantity | Total |
|---|---|---|---|---|
| Cell 1 | Cell 2 | Cell 2 | Cell 2 | Cell 2 |
| Cell 3 | Cell 3 | Cell 3 | Cell 3 | Cell 4 |

Subtotal: $10000

Table Options

| | |
|---|---|
| Responsive | Always |
| Borders | Default |
| Condensed | ○ |
| Striped | ○ |
| Dark | ○ |
| Hover | ○ |
| Show Table Header | ● |
| Show Caption | ○ |
| Show Table Footer | ○ |
| Unstyled | ○ |

```
</div>
▼ <div class="row">
  ▼ <div class="col">
    ▼ <div class="table-responsive">
```

Attributes

ID

Class Names: table-responsive

Key Value

```
element.style {
}

.table-responsive {
  overflow-x: auto;
  ~~-webkit-overflow-scrolling: touch;~~
}
                                    Bootstrap

div {
  display: block;
}
                                    styles.css
```

- Column
  - Table
    - Table Header
      - Table Row
        - Table Cell
        - Table Cell
        - Table Cell
        - Table Cell
        - Table Cell
    - Table Body

- Pages
  - cart.html
  - category.html
  - checkout.html
  - detail.html
  - index.html
  - login.html
  - Carousel-Hero.css

You will stop receiving updates soon!
Click here to renew your license.

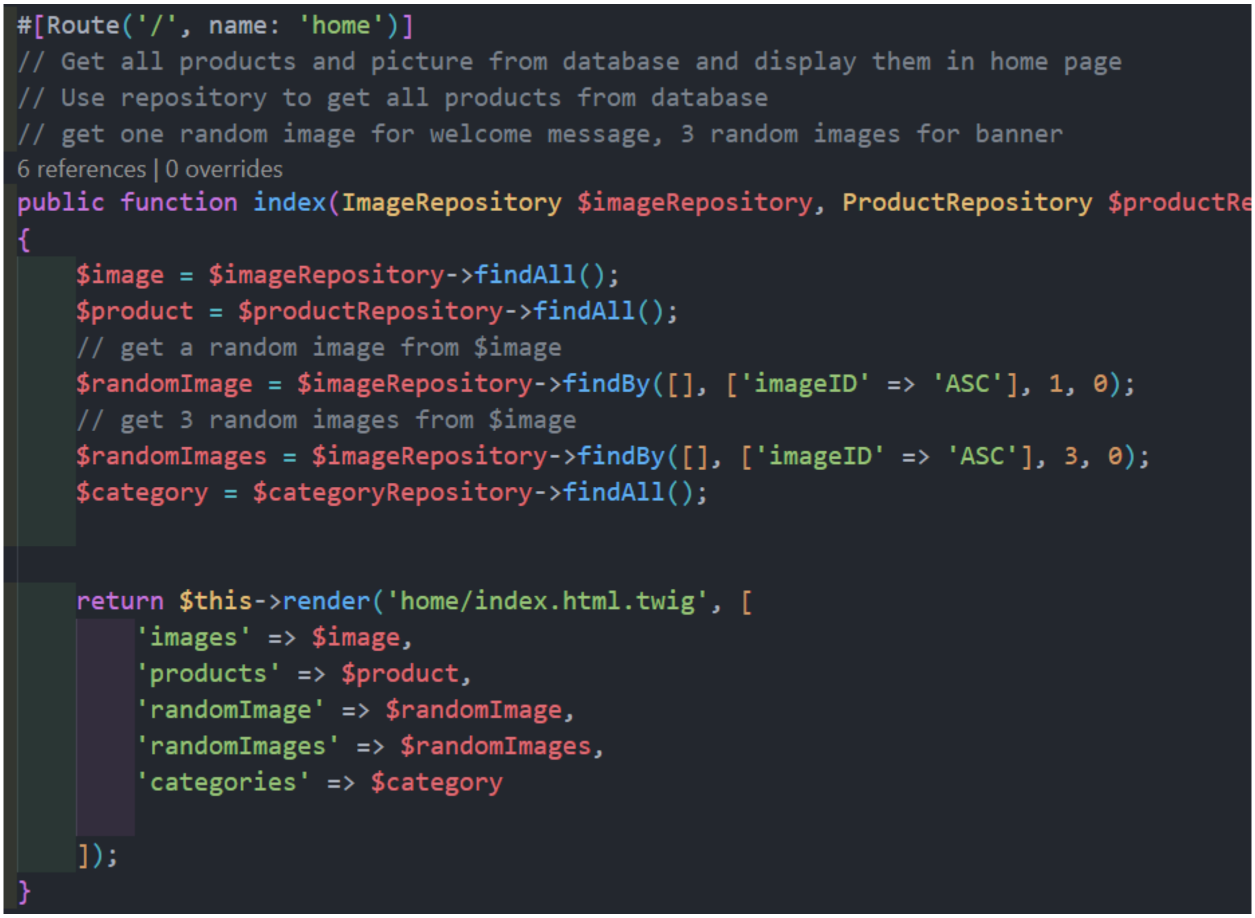
*Figure 12 Bootstrap Studio*

Thanks to the tags manager that is similar to layer management of design software like Photoshop, it becomes much easier and more convenient to nest and organizes components of the website. Furthermore, it presents the user with every component of CSS and Bootstrap library so the developers do not have to memorize every syntax and can focus on designing a good-looking website.

## 1.4. Apply to the project

### *1.4.1.Homepage*

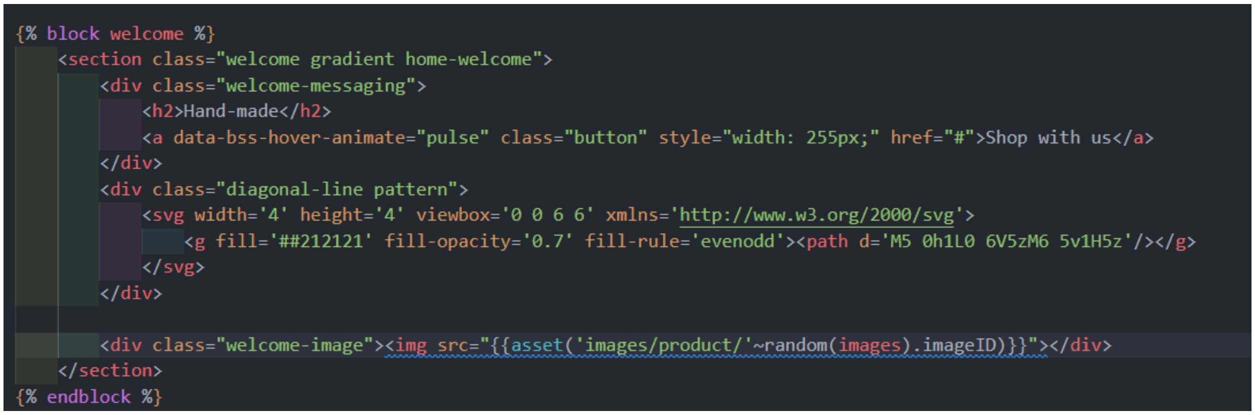
When the user opens the online store, they are greeted with an index of all the products in the store. This is done by the Controller taking the information of products and images on the database and passes it to the View for it to render out to the user.



```
#[Route('/', name: 'home')]
// Get all products and picture from database and display them in home page
// Use repository to get all products from database
// get one random image for welcome message, 3 random images for banner
6 references | 0 overrides
public function index(ImageRepository $imageRepository, ProductRepository $productRe
{
    $image = $imageRepository->findAll();
    $product = $productRepository->findAll();
    // get a random image from $image
    $randomImage = $imageRepository->findBy([], ['imageID' => 'ASC'], 1, 0);
    // get 3 random images from $image
    $randomImages = $imageRepository->findBy([], ['imageID' => 'ASC'], 3, 0);
    $category = $categoryRepository->findAll();


    return $this->render('home/index.html.twig', [
        'images' => $image,
        'products' => $product,
        'randomImage' => $randomImage,
        'randomImages' => $randomImages,
        'categories' => $category

    ]);
}
```

*Figure 13 Homepage Controller*

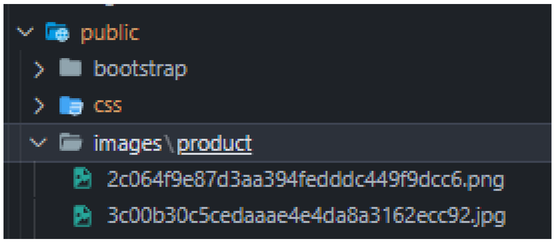
The user is first presented with a welcome message along with a random product image in the background



```
{% block welcome %}
    <section class="welcome gradient home-welcome">
        <div class="welcome-messaging">
            <h2>Hand-made</h2>
            <a data-bss-hover-animate="pulse" class="button" style="width: 255px;" href="#">Shop with us</a>
        </div>
        <div class="diagonal-line pattern">
            <svg width='4' height='4' viewbox='0 0 6 6' xmlns='http://www.w3.org/2000/svg'>
                <g fill='##212121' fill-opacity='0.7' fill-rule='evenodd'><path d='M5 0h1L0 6V5zM6 5v1H5z'/></g>
            </svg>
        </div>

        <div class="welcome-image"><img src="{{asset('images/product/'~random(images).imageID)}}"></div>
    </section>
{% endblock %}
```

*Figure 14 Welcome message*

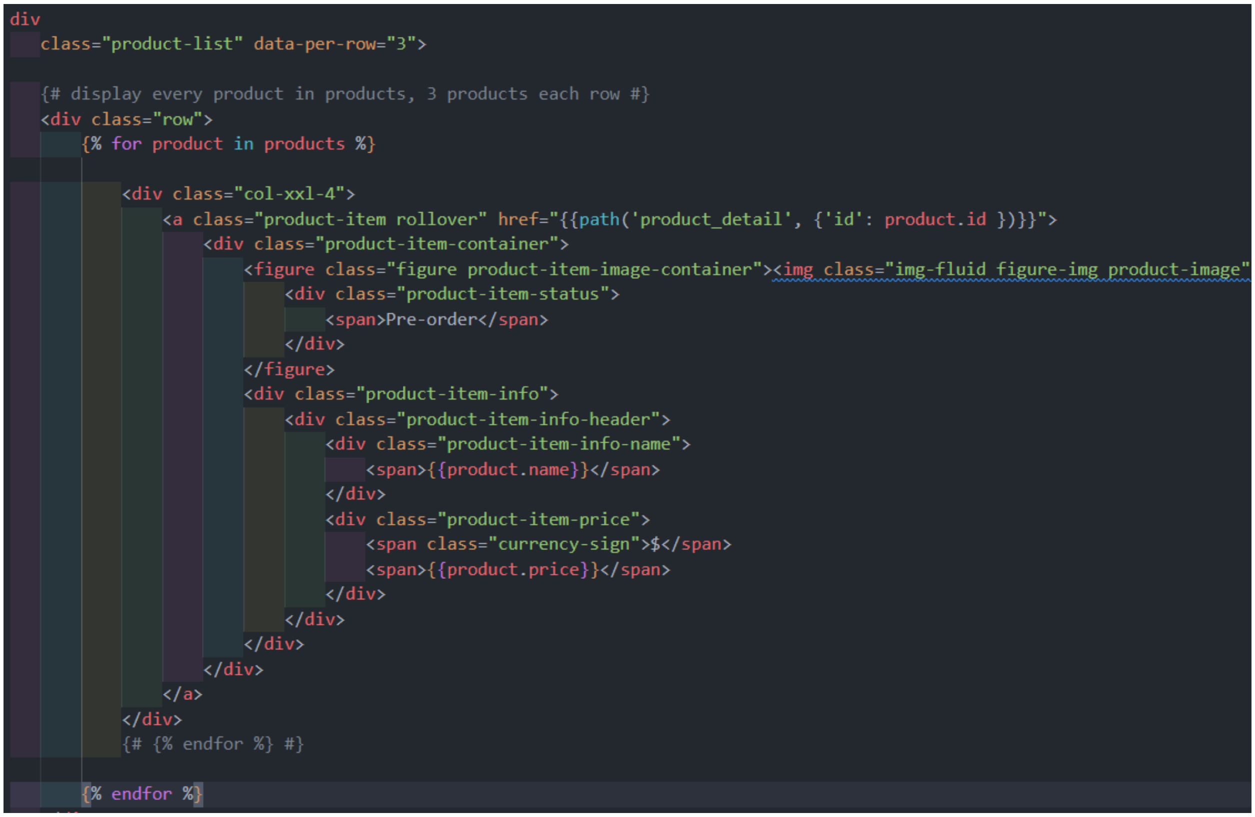
The “asset” means that it lies in the “public” folder of the project, “random(images)” is used to pick up a random image and “imageID” is the name of the image.



```
public
  bootstrap
  css
  images\product
    2c064f9e87d3aa394fedddc449f9dcc6.png
    3c00b30c5cedaaae4e4da8a3162ecc92.jpg
```

*Figure 15 Location of the images*

To display all of the product, a “foreach” Twig statement is used to go through every product in the database and display the necessary information.

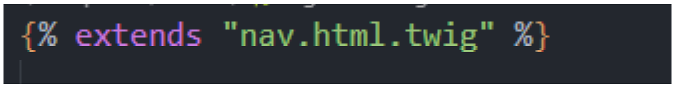


```
div
  class="product-list" data-per-row="3">

  {# display every product in products, 3 products each row #}
  <div class="row">
    {% for product in products %}

      <div class="col-xxl-4">
        <a class="product-item rollover" href="{{path('product_detail', {'id': product.id })}}">
          <div class="product-item-container">
            <figure class="figure product-item-image-container"><img class="img-fluid figure-img product-image"
              <div class="product-item-status">
                <span>Pre-order</span>
              </div>
            </figure>
            <div class="product-item-info">
              <div class="product-item-info-header">
                <div class="product-item-info-name">
                  <span>{{product.name}}</span>
                </div>
                <div class="product-item-price">
                  <span class="currency-sign">$</span>
                  <span>{{product.price}}</span>
                </div>
              </div>
            </div>
          </div>
        </a>
      </div>
      {# {% endfor %} #}

    {% endfor %}
```

*Figure 16 Display products*

### 1.4.2.Navigation bar

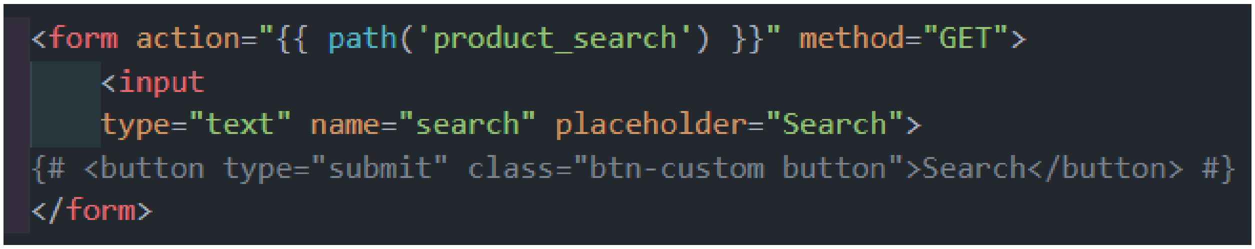
Navigation bar is an important part of this website, it appears on every page so that the customers along with admin can navigate through the store with ease. Since this component appears on every page, it is placed on its own file and is extended to by every View file



```
{% extends "nav.html.twig" %}
```

*Figure 17 Extends to navigation bar*

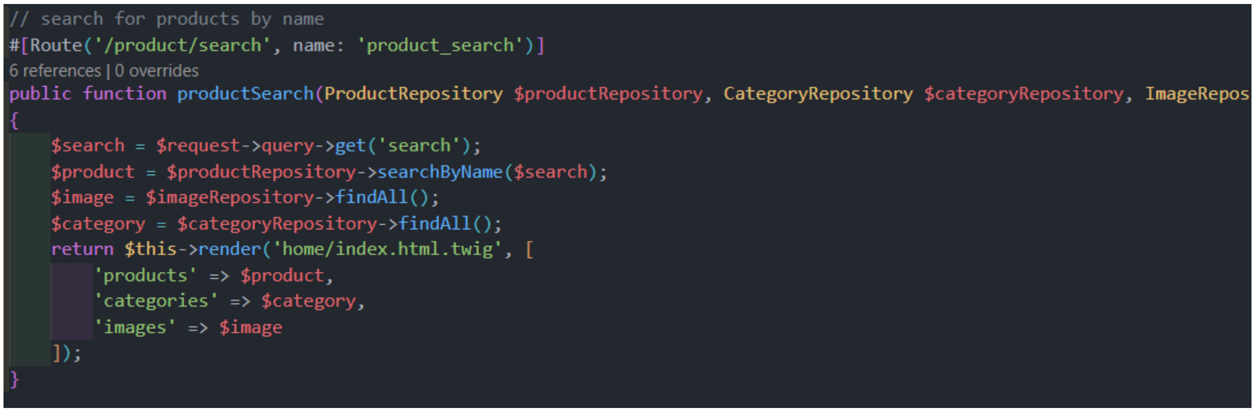
The navigation bar starts with a form which is a text field that let the user search for products by their name



```
<form action="{{ path('product_search') }}" method="GET">
        <input
        type="text" name="search" placeholder="Search">
{# <button type="submit" class="btn-custom button">Search</button> #}
</form>
```

*Figure 18 Search in nav bar*

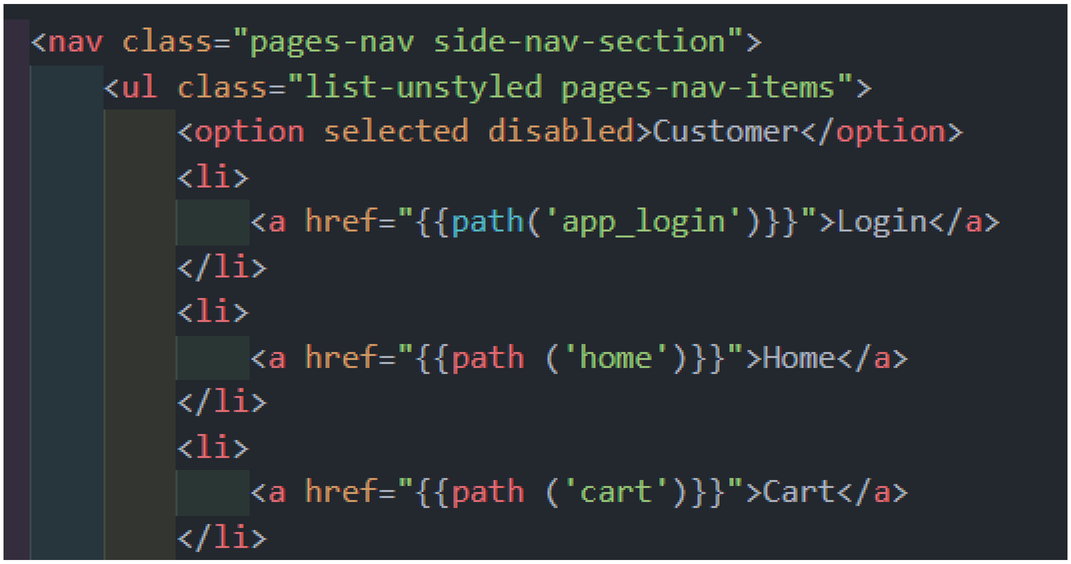
When the user submits the form, the view calls for a route named 'product_search' and pass the value that the user just input to the controller. From here, the controller searches the database to find products whose name matches or is similar to the queried name and render the homepage again but with the matching search results rather than every product.



```
// search for products by name
#[Route('/product/search', name: 'product_search')]
6 references | 0 overrides
public function productSearch(ProductRepository $productRepository, CategoryRepository $categoryRepository, ImageRepos
{
    $search = $request->query->get('search');
    $product = $productRepository->searchByName($search);
    $image = $imageRepository->findAll();
    $category = $categoryRepository->findAll();
    return $this->render('home/index.html.twig', [
        'products' => $product,
        'categories' => $category,
        'images' => $image
    ]);
}
```

*Figure 19 Search Controller*

After the search form, there are links that the user can use to go to certain sites of the store.



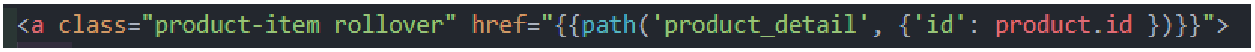
```
<nav class="pages-nav side-nav-section">
    <ul class="list-unstyled pages-nav-items">
        <option selected disabled>Customer</option>
        <li>
            <a href="{{path('app_login')}}">Login</a>
        </li>
        <li>
            <a href="{{path ('home')}}">Home</a>
        </li>
        <li>
            <a href="{{path ('cart')}}">Cart</a>
        </li>
```

*Figure 20 Navigation item*

Additionally, if the user is logged in with an admin account, extra items appear on the navigation bar, allowing them to perform CRUD actions on various items on the database.

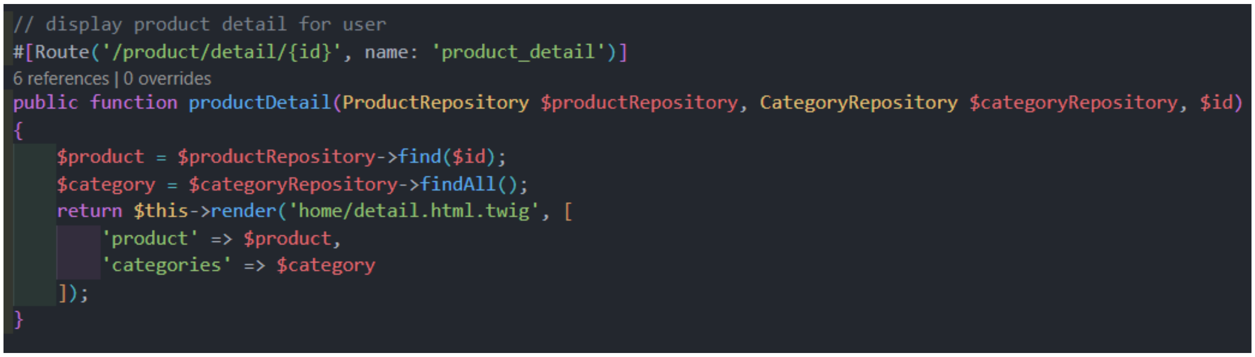
### 1.4.3. Product detail page

When the user clicks on a product on the index page, the View component sends a request to the Controller for the route 'product_detail' with the id of 'product.id' which is the id of the product clicked on



```
<a class="product-item rollover" href="{{path('product_detail', {'id': product.id })}}">
```

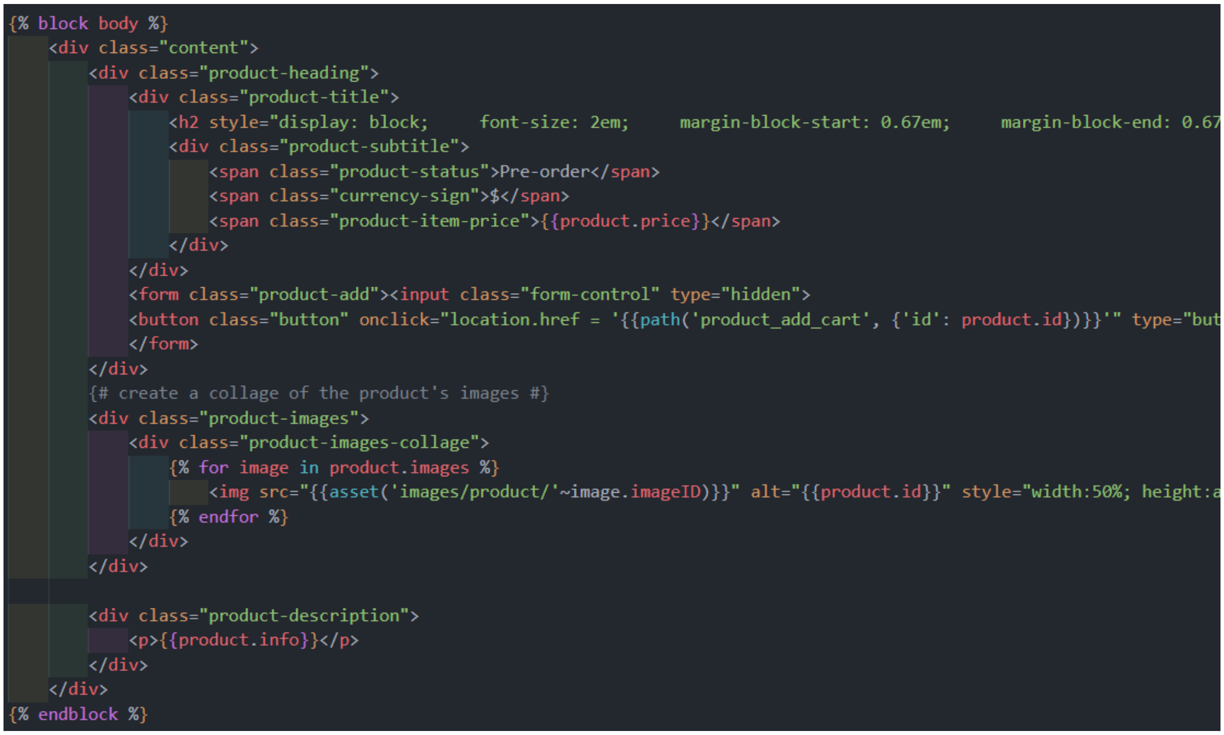
*Figure 21 Product detail link*



```
// display product detail for user
#[Route('/product/detail/{id}', name: 'product_detail')]
6 references | 0 overrides
public function productDetail(ProductRepository $productRepository, CategoryRepository $categoryRepository, $id)
{
    $product = $productRepository->find($id);
    $category = $categoryRepository->findAll();
    return $this->render('home/detail.html.twig', [
        'product' => $product,
        'categories' => $category
    ]);
}
```

*Figure 22 Product detail controller*

The Controller takes in the id of the product, searches the database for a match, then renders the detail.html.twig file with results returned from the database so the View can display the product's information.



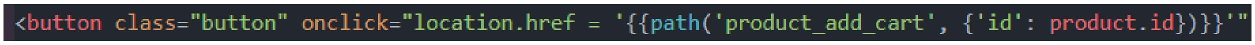
```
{% block body %}
    <div class="content">
        <div class="product-heading">
            <div class="product-title">
                <h2 style="display: block;     font-size: 2em;     margin-block-start: 0.67em;     margin-block-end: 0.67
                <div class="product-subtitle">
                    <span class="product-status">Pre-order</span>
                    <span class="currency-sign">$</span>
                    <span class="product-item-price">{{product.price}}</span>
                </div>
            </div>
            <form class="product-add"><input class="form-control" type="hidden">
            <button class="button" onclick="location.href = '{{path('product_add_cart', {'id': product.id})}}'" type="but
            </form>
        </div>
        {# create a collage of the product's images #}
        <div class="product-images">
            <div class="product-images-collage">
                {% for image in product.images %}
                    <img src="{{asset('images/product/'~image.imageID)}}" alt="{{product.id}}" style="width:50%; height:a
                {% endfor %}
            </div>
        </div>

        <div class="product-description">
            <p>{{product.info}}</p>
        </div>
    </div>
{% endblock %}
```

*Figure 23 View of product detail*

The information displayed for the product includes product’s name, product’s price, product’s image(s), product’s description, and a “Add to cart” button.

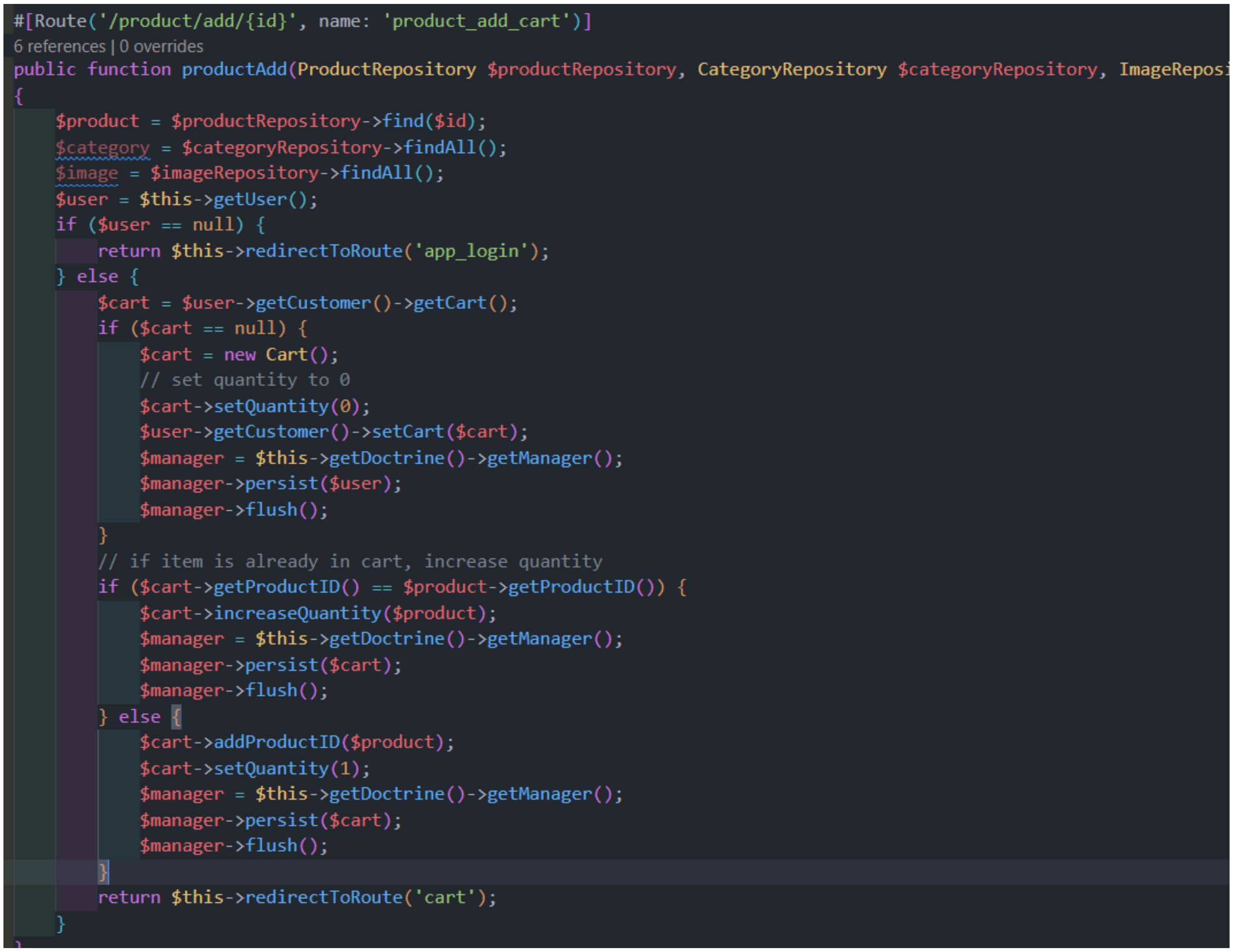
### 1.4.4.Cart page

When the user clicks on the “Add to cart” button, logged in user is taken to the cart page with the item they just clicked the button on appearing in it. If the user has not logged in, the website redirects the user to the login page so they can login before adding an item into their cart.



```
<button class="button" onclick="location.href = '{{path('product_add_cart', {'id': product.id})}}'"
```

*Figure 24 Add to cart button*

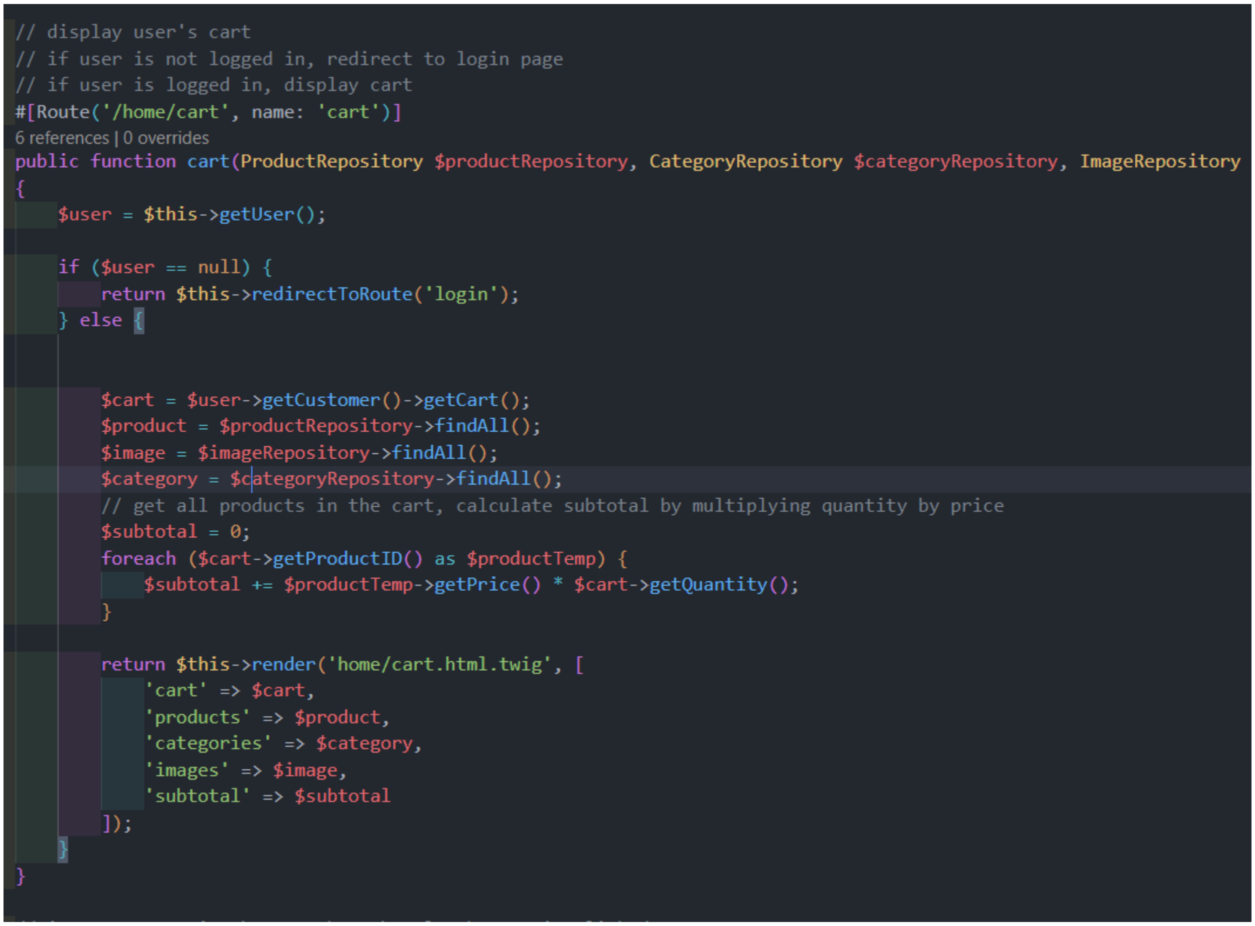


```
#[Route('/product/add/{id}', name: 'product_add_cart')]
6 references | 0 overrides
public function productAdd(ProductRepository $productRepository, CategoryRepository $categoryRepository, ImageReposi
{
    $product = $productRepository->find($id);
    $category = $categoryRepository->findAll();
    $image = $imageRepository->findAll();
    $user = $this->getUser();
    if ($user == null) {
        return $this->redirectToRoute('app_login');
    } else {
        $cart = $user->getCustomer()->getCart();
        if ($cart == null) {
            $cart = new Cart();
            // set quantity to 0
            $cart->setQuantity(0);
            $user->getCustomer()->setCart($cart);
            $manager = $this->getDoctrine()->getManager();
            $manager->persist($user);
            $manager->flush();
        }
        // if item is already in cart, increase quantity
        if ($cart->getProductID() == $product->getProductID()) {
            $cart->increaseQuantity($product);
            $manager = $this->getDoctrine()->getManager();
            $manager->persist($cart);
            $manager->flush();
        } else {
            $cart->addProductID($product);
            $cart->setQuantity(1);
            $manager = $this->getDoctrine()->getManager();
            $manager->persist($cart);
            $manager->flush();
        }
        return $this->redirectToRoute('cart');
    }
}
```

*Figure 25 Add to Cart Controller*

The Controller takes in an id in order to find the product in the database. Then it checks if the user has logged into the website, if not, the user is directed to the login page. If the user is logged in, the system checks if the user has a Cart in their account yet, if not then the Controller creates a cart for the account, allowing the items to be added to the cart. Then the Controller checks if the product is already in the cart, if it is not, then the item is added to the cart, if it is, then the quantity of that product is increased by 1.

After the user has added an item to the cart or clicked on the “cart” link on the navigation bar, they are taken to the cart site.

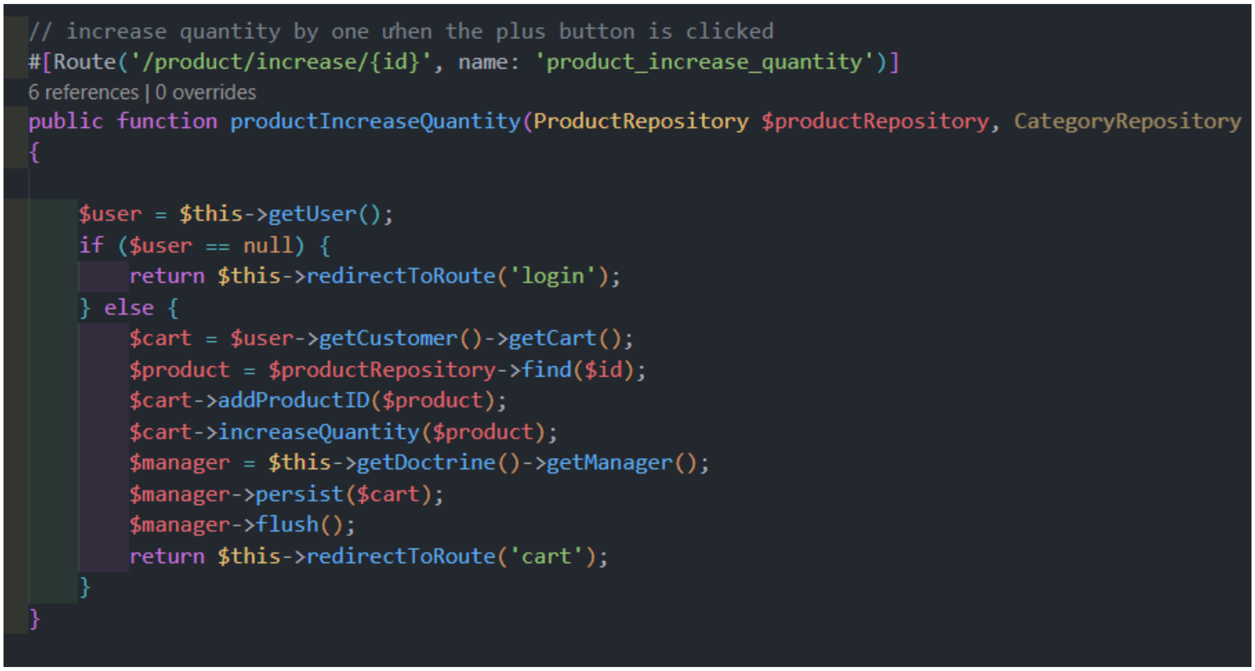


```
// display user's cart
// if user is not logged in, redirect to login page
// if user is logged in, display cart
#[Route('/home/cart', name: 'cart')]
6 references | 0 overrides
public function cart(ProductRepository $productRepository, CategoryRepository $categoryRepository, ImageRepository
{
    $user = $this->getUser();

    if ($user == null) {
        return $this->redirectToRoute('login');
    } else {


        $cart = $user->getCustomer()->getCart();
        $product = $productRepository->findAll();
        $image = $imageRepository->findAll();
        $category = $categoryRepository->findAll();
        // get all products in the cart, calculate subtotal by multiplying quantity by price
        $subtotal = 0;
        foreach ($cart->getProductID() as $productTemp) {
            $subtotal += $productTemp->getPrice() * $cart->getQuantity();
        }

        return $this->render('home/cart.html.twig', [
            'cart' => $cart,
            'products' => $product,
            'categories' => $category,
            'images' => $image,
            'subtotal' => $subtotal
        ]);
    }
}
```

*Figure 26 Cart Controller*

If the user has not logged into the website, they are redirected to the login site. Else the Controller takes in the data of the user’s cart along with product, images, and categories from the database. It also declares a subtotal variable to calculate the subtotal of the cart by going through every item in the cart multiplying unit price and quantity before adding it to the subtotal. Finally, it renders the file cart.html.twig to the user passing all of the variables.

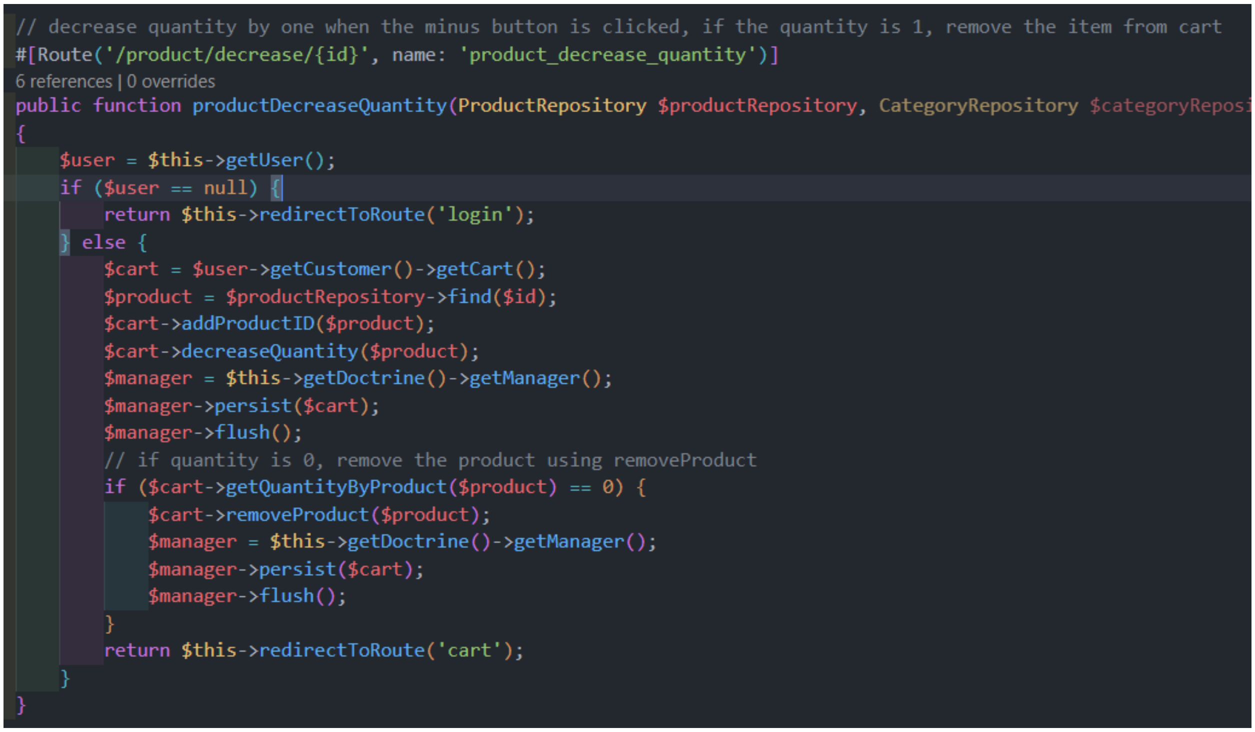
In the cart page, there is an increase and decrease quantity button for the user to change the quantity of the products they want to buy.



```php
// increase quantity by one when the plus button is clicked
#[Route('/product/increase/{id}', name: 'product_increase_quantity')]
6 references | 0 overrides
public function productIncreaseQuantity(ProductRepository $productRepository, CategoryRepository
{

    $user = $this->getUser();
    if ($user == null) {
        return $this->redirectToRoute('login');
    } else {
        $cart = $user->getCustomer()->getCart();
        $product = $productRepository->find($id);
        $cart->addProductID($product);
        $cart->increaseQuantity($product);
        $manager = $this->getDoctrine()->getManager();
        $manager->persist($cart);
        $manager->flush();
        return $this->redirectToRoute('cart');
    }
}
```

*Figure 27 Increase quantity*

The Controller takes in the item in the cart, increases its quantity by one, then push it back into the database and render the cart page to the user.

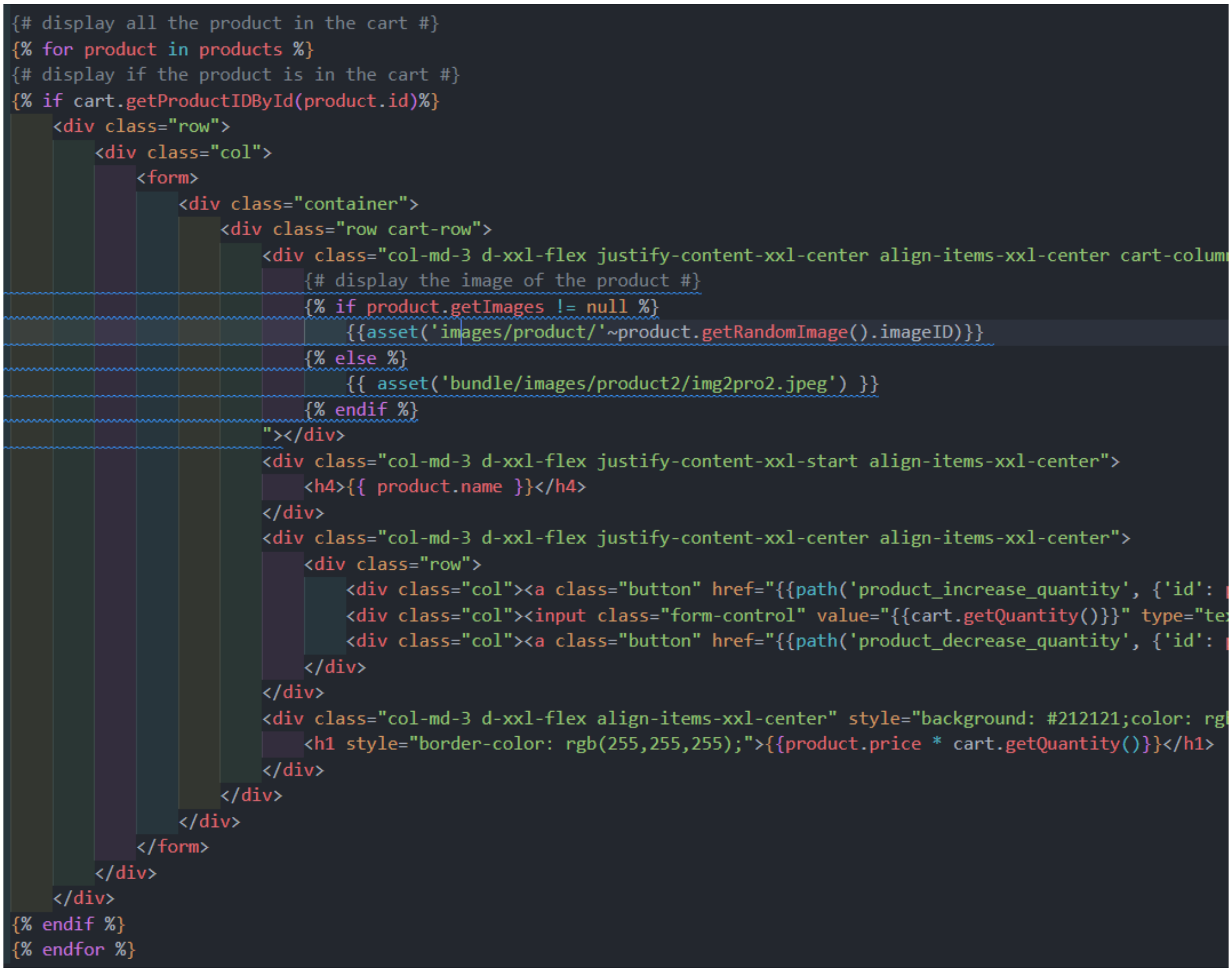


```php
// decrease quantity by one when the minus button is clicked, if the quantity is 1, remove the item from cart
#[Route('/product/decrease/{id}', name: 'product_decrease_quantity')]
6 references | 0 overrides
public function productDecreaseQuantity(ProductRepository $productRepository, CategoryRepository $categoryReposi
{
    $user = $this->getUser();
    if ($user == null) {
        return $this->redirectToRoute('login');
    } else {
        $cart = $user->getCustomer()->getCart();
        $product = $productRepository->find($id);
        $cart->addProductID($product);
        $cart->decreaseQuantity($product);
        $manager = $this->getDoctrine()->getManager();
        $manager->persist($cart);
        $manager->flush();
        // if quantity is 0, remove the product using removeProduct
        if ($cart->getQuantityByProduct($product) == 0) {
            $cart->removeProduct($product);
            $manager = $this->getDoctrine()->getManager();
            $manager->persist($cart);
            $manager->flush();
        }
        return $this->redirectToRoute('cart');
    }
}
```

*Figure 28 Decrease quantity*

The decrease quantity function works similarly to the increase counterpart, with a bonus that if the quantity of the product falls to 0, it is removed from the cart.

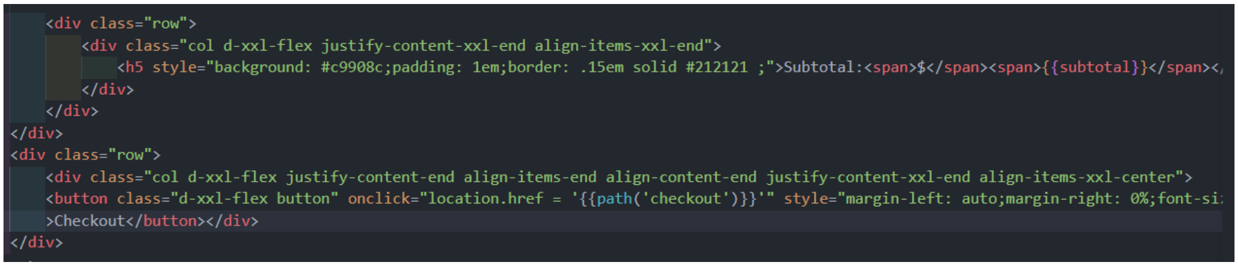
In the cart View, each of the product has its image, name, quantity, unitprice * quantity displayed



```
{# display all the product in the cart #}
{% for product in products %}
{# display if the product is in the cart #}
{% if cart.getProductIDById(product.id)%}
    <div class="row">
        <div class="col">
            <form>
                <div class="container">
                    <div class="row cart-row">
                        <div class="col-md-3 d-xxl-flex justify-content-xxl-center align-items-xxl-center cart-colum
                            {# display the image of the product #}
                            {% if product.getImages != null %}
                                {{asset('images/product/'~product.getRandomImage().imageID)}}
                            {% else %}
                                {{ asset('bundle/images/product2/img2pro2.jpeg') }}
                            {% endif %}
                        "></div>
                        <div class="col-md-3 d-xxl-flex justify-content-xxl-start align-items-xxl-center">
                            <h4>{{ product.name }}</h4>
                        </div>
                        <div class="col-md-3 d-xxl-flex justify-content-xxl-center align-items-xxl-center">
                            <div class="row">
                                <div class="col"><a class="button" href="{{path('product_increase_quantity', {'id':
                                <div class="col"><input class="form-control" value="{{cart.getQuantity()}}" type="te
                                <div class="col"><a class="button" href="{{path('product_decrease_quantity', {'id':
                            </div>
                        </div>
                        <div class="col-md-3 d-xxl-flex align-items-xxl-center" style="background: #212121;color: rg
                            <h1 style="border-color: rgb(255,255,255);">{{product.price * cart.getQuantity()}}</h1>
                        </div>
                    </div>
                </div>
            </form>
        </div>
    </div>
{% endif %}
{% endfor %}
```

*Figure 29 Cart View*

Under the cart, the subtotal and a “Checkout” button is displayed so the user can checkout with the items in the cart.

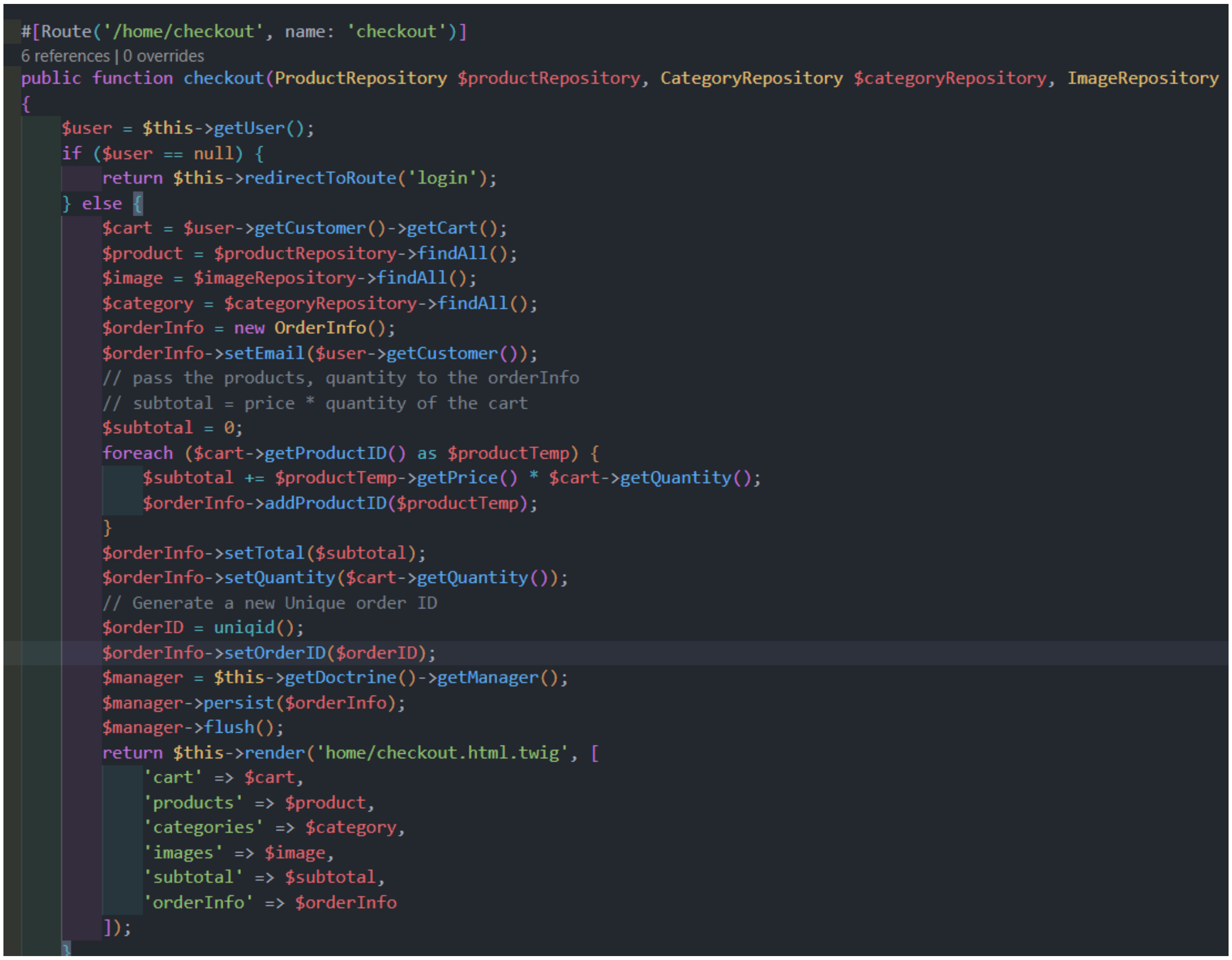


```
    <div class="row">
        <div class="col d-xxl-flex justify-content-xxl-end align-items-xxl-end">
            <h5 style="background: #c9908c;padding: 1em;border: .15em solid #212121 ;">Subtotal:<span>$</span><span>{{subtotal}}</span><
        </div>
    </div>
</div>
<div class="row">
    <div class="col d-xxl-flex justify-content-end align-items-end align-content-end justify-content-xxl-end align-items-xxl-center">
    <button class="d-xxl-flex button" onclick="location.href = '{{path('checkout')}}'" style="margin-left: auto;margin-right: 0%;font-si
    >Checkout</button></div>
</div>
```

*Figure 30 Subtotal and Checkout on Cart View*

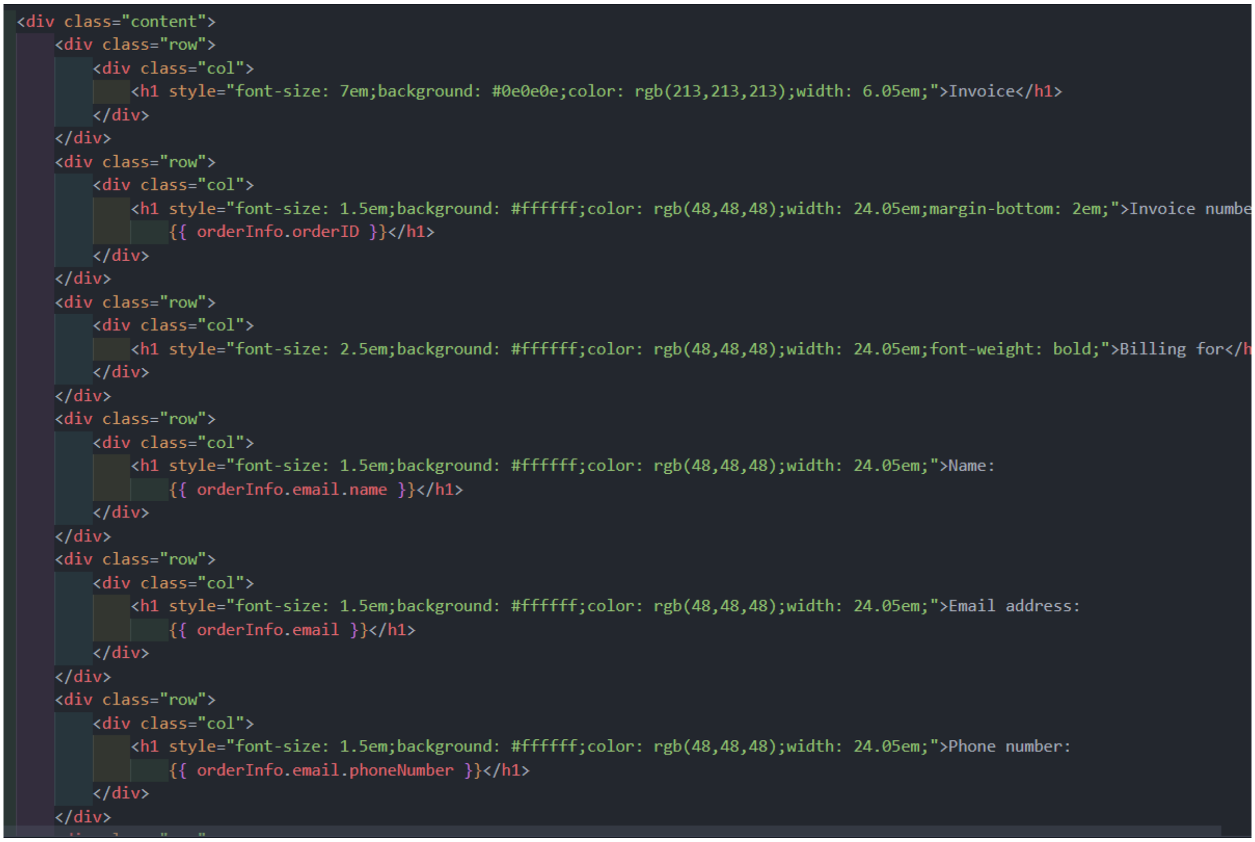
### 1.4.5.Checkout

When a customer clicks on the “Checkout” button, a request for checkout is sent to the Controller. At this stage, all of the items in the cart are transferred into an invoice and the items are cleared from the cart. After the items are transferred, customer’s information is also copied into the invoice. Finally an orderID is generated using uniqid() before the invoice is created in the database. After an invoice is created, checkout.html.twig is rendered to display the invoice to the user.



```
#[Route('/home/checkout', name: 'checkout')]
6 references | 0 overrides
public function checkout(ProductRepository $productRepository, CategoryRepository $categoryRepository, ImageRepository
{
    $user = $this->getUser();
    if ($user == null) {
        return $this->redirectToRoute('login');
    } else {
        $cart = $user->getCustomer()->getCart();
        $product = $productRepository->findAll();
        $image = $imageRepository->findAll();
        $category = $categoryRepository->findAll();
        $orderInfo = new OrderInfo();
        $orderInfo->setEmail($user->getCustomer());
        // pass the products, quantity to the orderInfo
        // subtotal = price * quantity of the cart
        $subtotal = 0;
        foreach ($cart->getProductID() as $productTemp) {
            $subtotal += $productTemp->getPrice() * $cart->getQuantity();
            $orderInfo->addProductID($productTemp);
        }
        $orderInfo->setTotal($subtotal);
        $orderInfo->setQuantity($cart->getQuantity());
        // Generate a new Unique order ID
        $orderID = uniqid();
        $orderInfo->setOrderID($orderID);
        $manager = $this->getDoctrine()->getManager();
        $manager->persist($orderInfo);
        $manager->flush();
        return $this->render('home/checkout.html.twig', [
            'cart' => $cart,
            'products' => $product,
            'categories' => $category,
            'images' => $image,
            'subtotal' => $subtotal,
            'orderInfo' => $orderInfo
        ]);
    }
```

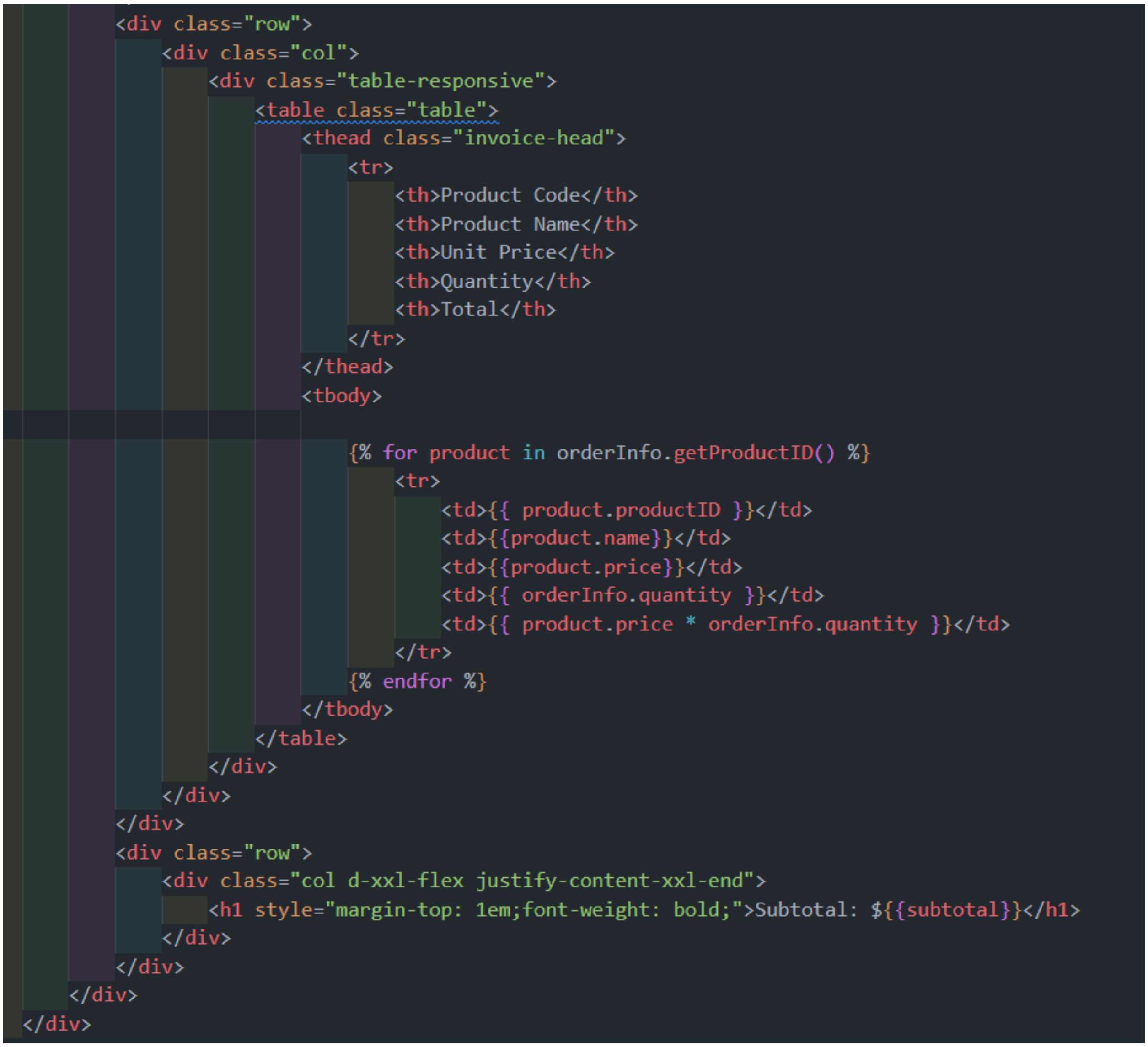
*Figure 31 Checkout Controller*



```
<div class="content">
    <div class="row">
        <div class="col">
            <h1 style="font-size: 7em;background: #0e0e0e;color: rgb(213,213,213);width: 6.05em;">Invoice</h1>
        </div>
    </div>
    <div class="row">
        <div class="col">
            <h1 style="font-size: 1.5em;background: #ffffff;color: rgb(48,48,48);width: 24.05em;margin-bottom: 2em;">Invoice numbe
                {{ orderInfo.orderID }}</h1>
        </div>
    </div>
    <div class="row">
        <div class="col">
            <h1 style="font-size: 2.5em;background: #ffffff;color: rgb(48,48,48);width: 24.05em;font-weight: bold;">Billing for</h
        </div>
    </div>
    <div class="row">
        <div class="col">
            <h1 style="font-size: 1.5em;background: #ffffff;color: rgb(48,48,48);width: 24.05em;">Name:
                {{ orderInfo.email.name }}</h1>
        </div>
    </div>
    <div class="row">
        <div class="col">
            <h1 style="font-size: 1.5em;background: #ffffff;color: rgb(48,48,48);width: 24.05em;">Email address:
                {{ orderInfo.email }}</h1>
        </div>
    </div>
    <div class="row">
        <div class="col">
            <h1 style="font-size: 1.5em;background: #ffffff;color: rgb(48,48,48);width: 24.05em;">Phone number:
                {{ orderInfo.email.phoneNumber }}</h1>
        </div>
    </div>
```

*Figure 32 Checkout View*

First, the invoice displays the orderID and the customer’s information including name, email address, and phone number. After that, the list of items in the invoice is displayed in a table including product code, name, quantity, unit price, and total price. After the table, there is a subtotal displaying the subtotal amount of the whole order.

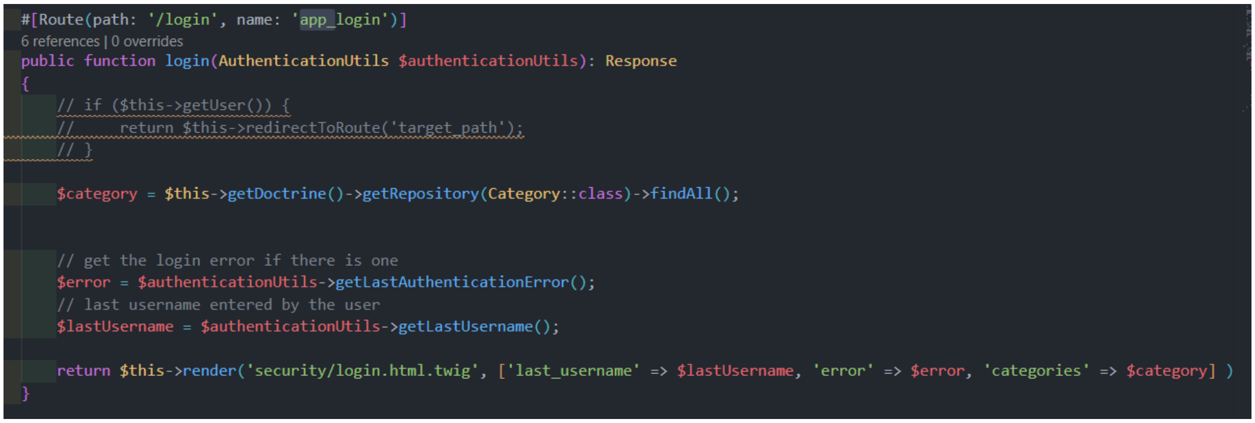


```
        <div class="row">
            <div class="col">
                <div class="table-responsive">
                    <table class="table">
                        <thead class="invoice-head">
                            <tr>
                                <th>Product Code</th>
                                <th>Product Name</th>
                                <th>Unit Price</th>
                                <th>Quantity</th>
                                <th>Total</th>
                            </tr>
                        </thead>
                        <tbody>

                            {% for product in orderInfo.getProductID() %}
                                <tr>
                                    <td>{{ product.productID }}</td>
                                    <td>{{product.name}}</td>
                                    <td>{{product.price}}</td>
                                    <td>{{ orderInfo.quantity }}</td>
                                    <td>{{ product.price * orderInfo.quantity }}</td>
                                </tr>
                            {% endfor %}
                        </tbody>
                    </table>
                </div>
            </div>
        </div>
        <div class="row">
            <div class="col d-xxl-flex justify-content-xxl-end">
                <h1 style="margin-top: 1em;font-weight: bold;">Subtotal: ${{subtotal}}</h1>
            </div>
        </div>
    </div>
</div>
```

*Figure 33 Checkout View*

### 1.4.6.Log in

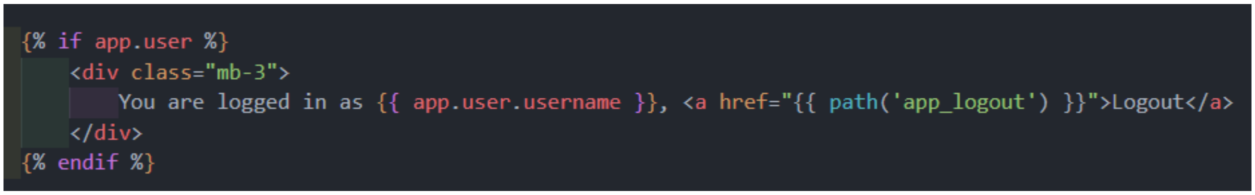
When a user navigates or is directed to the login page. The Controller finds if there is any user logged into the website and displays it if there is before presenting the user with a login form.



```
#[Route(path: '/login', name: 'app_login')]
6 references | 0 overrides
public function login(AuthenticationUtils $authenticationUtils): Response
{
    // if ($this->getUser()) {
    //     return $this->redirectToRoute('target_path');
    // }

    $category = $this->getDoctrine()->getRepository(Category::class)->findAll();


    // get the login error if there is one
    $error = $authenticationUtils->getLastAuthenticationError();
    // last username entered by the user
    $lastUsername = $authenticationUtils->getLastUsername();

    return $this->render('security/login.html.twig', ['last_username' => $lastUsername, 'error' => $error, 'categories' => $category] )
}
```

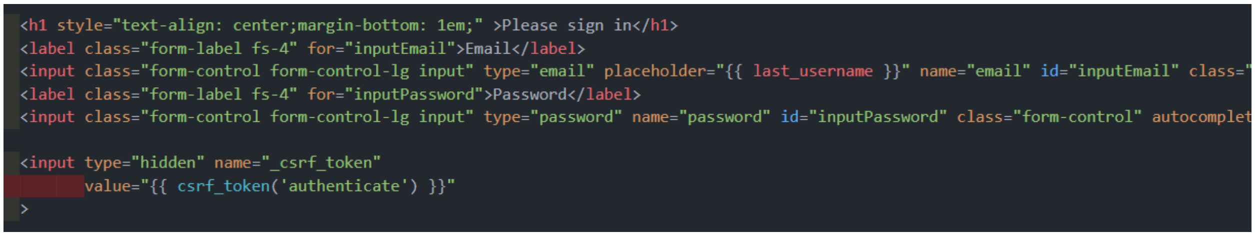
*Figure 34 Login Controller*

In the View, if a user is logged in, the website lets them know and give them the option to log out



```
{% if app.user %}
    <div class="mb-3">
        You are logged in as {{ app.user.username }}, <a href="{{ path('app_logout') }}">Logout</a>
    </div>
{% endif %}
```

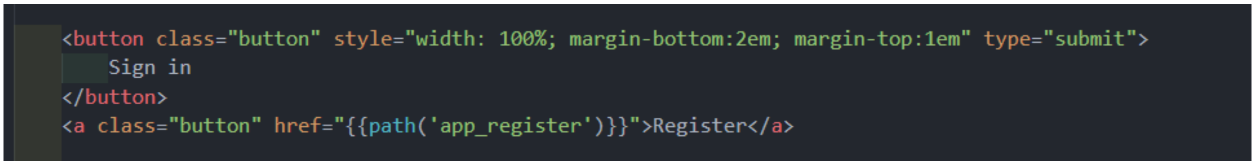
*Figure 35 Log in View*



```
<h1 style="text-align: center;margin-bottom: 1em;" >Please sign in</h1>
<label class="form-label fs-4" for="inputEmail">Email</label>
<input class="form-control form-control-lg input" type="email" placeholder="{{ last_username }}" name="email" id="inputEmail" class="
<label class="form-label fs-4" for="inputPassword">Password</label>
<input class="form-control form-control-lg input" type="password" name="password" id="inputPassword" class="form-control" autocomplet

<input type="hidden" name="_csrf_token"
       value="{{ csrf_token('authenticate') }}"
>
```

*Figure 36 Log in View*

After the option to logout, there is a form for the user to input the login credentials followed by sign in and register button.

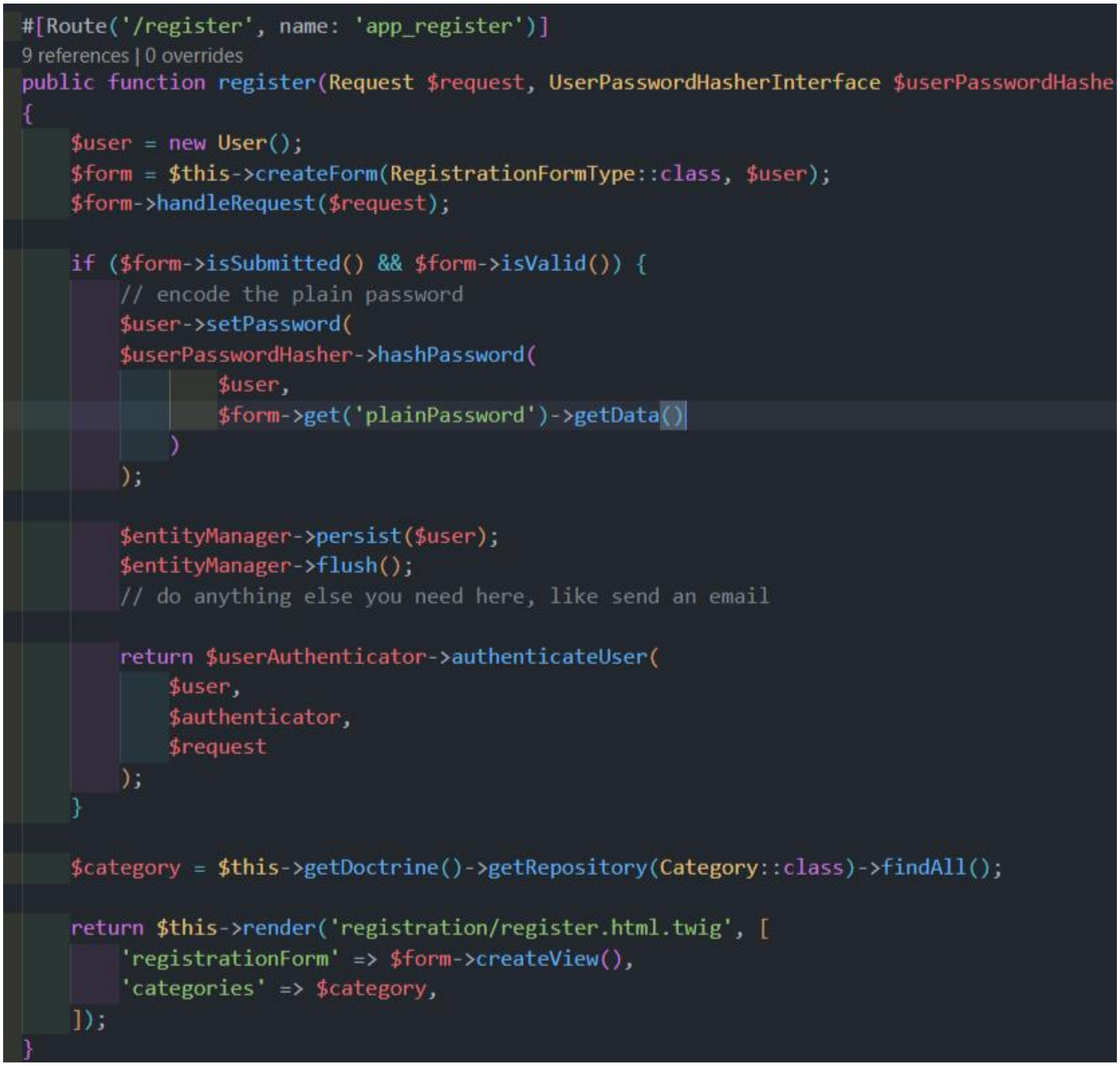


```
<button class="button" style="width: 100%; margin-bottom:2em; margin-top:1em" type="submit">
    Sign in
</button>
<a class="button" href="{{path('app_register')}}">Register</a>
```

*Figure 37 Log in View*

### 1.4.7.Register

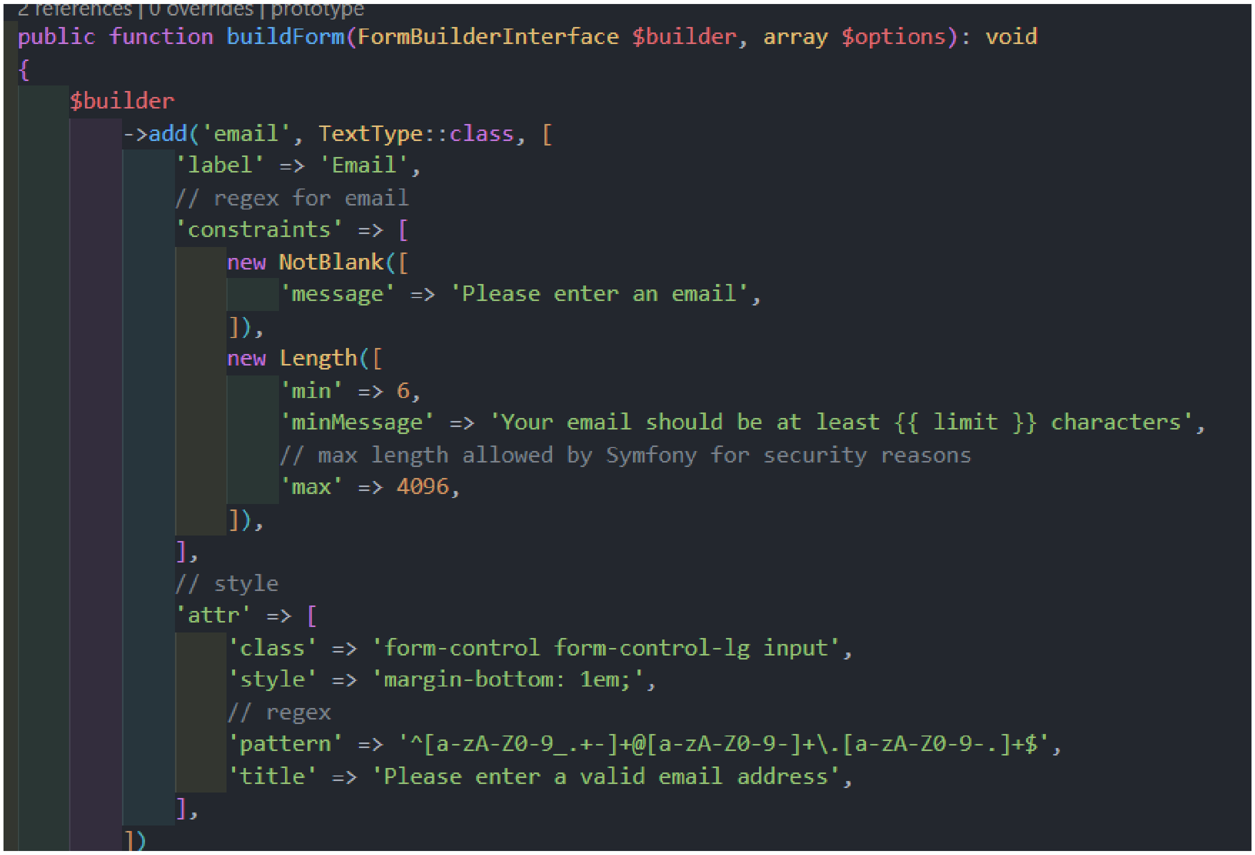
In the login page, the user has the option to click on the "Register" button and create a new account. When the "Register" button is clicked, the Controller builds a form for the user to input the account information that they want to create and then hash the password before creating a user in the database.



```
#[Route('/register', name: 'app_register')]
9 references | 0 overrides
public function register(Request $request, UserPasswordHasherInterface $userPasswordHashe
{
    $user = new User();
    $form = $this->createForm(RegistrationFormType::class, $user);
    $form->handleRequest($request);

    if ($form->isSubmitted() && $form->isValid()) {
        // encode the plain password
        $user->setPassword(
        $userPasswordHasher->hashPassword(
                $user,
                $form->get('plainPassword')->getData()
            )
        );

        $entityManager->persist($user);
        $entityManager->flush();
        // do anything else you need here, like send an email

        return $userAuthenticator->authenticateUser(
            $user,
            $authenticator,
            $request
        );
    }

    $category = $this->getDoctrine()->getRepository(Category::class)->findAll();

    return $this->render('registration/register.html.twig', [
        'registrationForm' => $form->createView(),
        'categories' => $category,
    ]);
}
```

*Figure 38 Registration Controller*

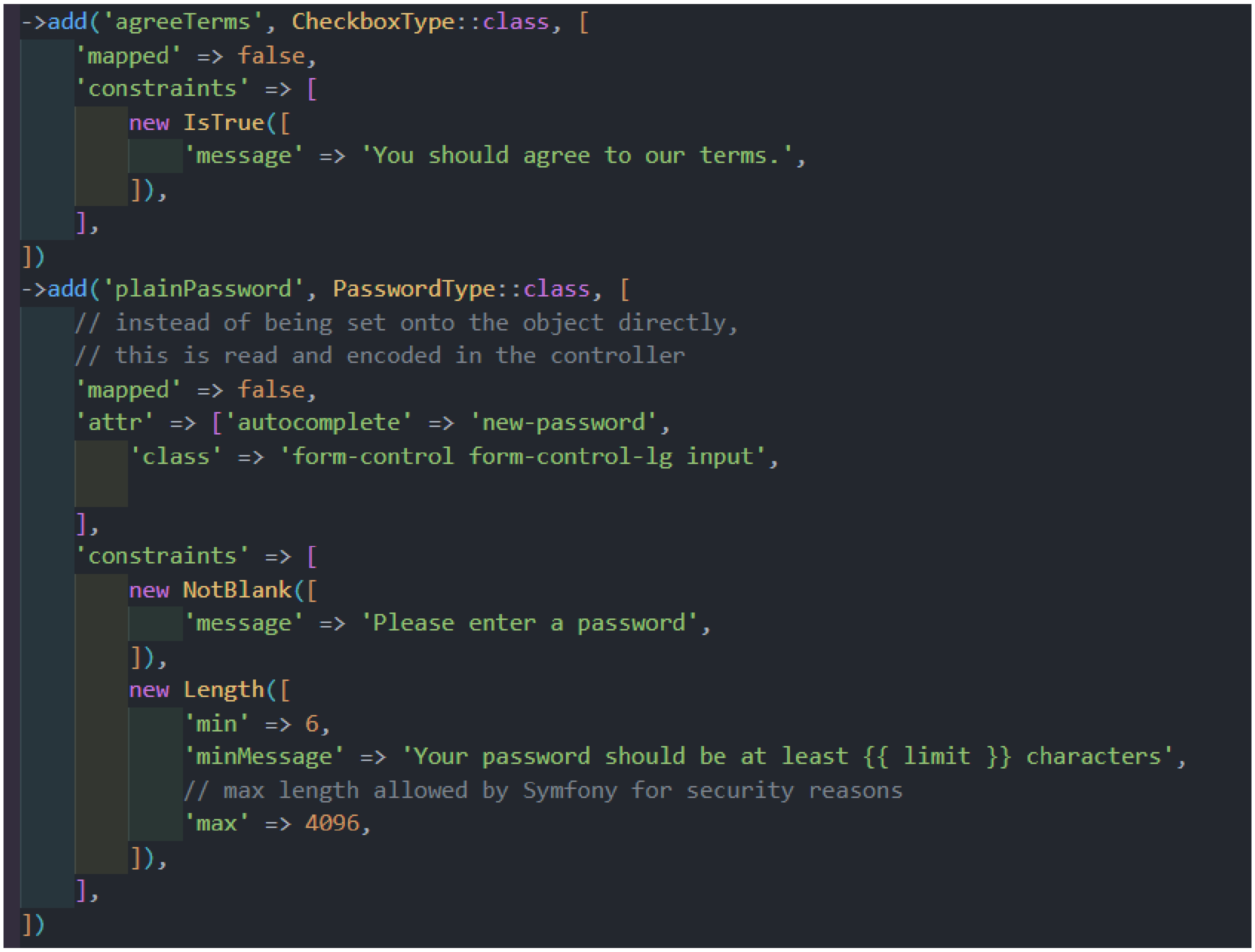
The form is constructed as the following



```
2 references | 0 overrides | prototype
public function buildForm(FormBuilderInterface $builder, array $options): void
{
    $builder
        ->add('email', TextType::class, [
            'label' => 'Email',
            // regex for email
            'constraints' => [
                new NotBlank([
                    'message' => 'Please enter an email',
                ]),
                new Length([
                    'min' => 6,
                    'minMessage' => 'Your email should be at least {{ limit }} characters',
                    // max length allowed by Symfony for security reasons
                    'max' => 4096,
                ]),
            ],
            // style
            'attr' => [
                'class' => 'form-control form-control-lg input',
                'style' => 'margin-bottom: 1em;',
                // regex
                'pattern' => '^[a-zA-Z0-9_.+-]+@[a-zA-Z0-9-]+\.[a-zA-Z0-9-.]+$',
                'title' => 'Please enter a valid email address',
            ],
        ])
```

*Figure 39 Register Form*

First, there is a field for email. In this field, the user cannot leave blank, the minimum length is 6 characters and maximum is 4096. The input must follow conventional email pattern of name@domain.com to proceed.



```
->add('agreeTerms', CheckboxType::class, [
    'mapped' => false,
    'constraints' => [
        new IsTrue([
            'message' => 'You should agree to our terms.',
        ]),
    ],
])
->add('plainPassword', PasswordType::class, [
    // instead of being set onto the object directly,
    // this is read and encoded in the controller
    'mapped' => false,
    'attr' => ['autocomplete' => 'new-password',
        'class' => 'form-control form-control-lg input',

    ],
    'constraints' => [
        new NotBlank([
            'message' => 'Please enter a password',
        ]),
        new Length([
            'min' => 6,
            'minMessage' => 'Your password should be at least {{ limit }} characters',
            // max length allowed by Symfony for security reasons
            'max' => 4096,
        ]),
    ],
])
```

*Figure 40 Registration form*

Then there is an agree term checkbox which indicates that the user has accepted the terms of services of the website, the user has to check this box to register. Finally, there is plain password. This is the input where the user type in the raw password of their account before being hashed and encrypted. The field cannot be blank and must have from 6 to 4096 characters.

## 2. How to develop a Symfony project

Assuming that one has already installed Composer and Symfony, here are the step needed to develop a Symfony project:

2.1. Open the terminal and run the command “symfony new --full <project name>”
2.2. Configurate the .env file so that Symfony can create a new database.
2.3. Create a new database for the project with the command “php bin/console doctrine:database:create”
2.4. Install security bundle so that there can be User and login/sign up feature with the command “composer require symfony/security-bundle”
2.5. Create user entity with “php bin/console make:user”
2.6. Create authentication with “php bin/console make:auth”
2.7. Modify the file src/Security/UserAuthenticator.php to redirect the user once they logs in successfully

2.8. Create a registration form with “php bin/console make:registration-form”

2.9. Create entities with “php bin/console make:entity”

2.10. Create migration and migrate the entity onto the database with “php bin/console make:migration” then “php bin/console doctrine:migrations:migrate”

2.11. Create roles by editing the file config/packages/security.yaml under the “access_control” section

2.12. Create controllers for entities using “php bin/console make:controller”

2.13. Create forms for entities using ““php bin/console make:form”

2.14. Create file src/Security/AccessDeniedHandler.php to handle access denied event

2.15. Configurate the file config/packages/security.yaml so that the accessdeniedhandler file can be ran.

2.16. Edit the Controller, Form and View files as needed and run “Symfony:serve” to start running a Symfony project.

## 3. Extra important code

### 3.1. .env

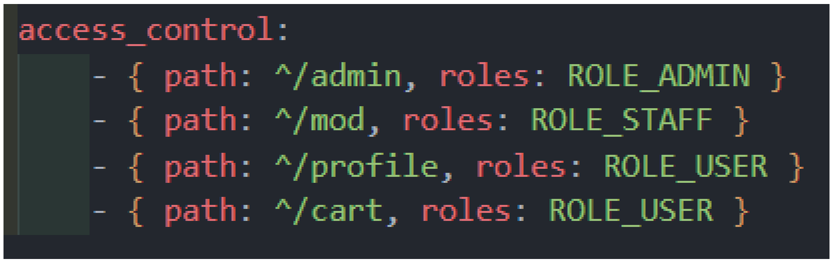
This file needs to be edited so that Symfony knows how to create a new database. In this project, database is managed by XAMPP, so the line 31 needs to be commented and line 30 needs to be uncommented and edited to <DATABASE_URL="mysql://root:@127.0.0.1:3306/ShoppingCartDB">

### 3.2. Config/services.yaml

In this file, the “parameter” section needs to be edited so that Symfony knows where to store pictures uploaded by the admin. Under the “parameter:”, the line need to be added is “product_image : '%kernel.project_dir%/public/images/product'”.

### 3.3. Config/packages/security.yaml

In this file, access_control need to be modified so that only certain roles of the system can go to certain sites. For example, only Admin users can go to pages with /admin. If someone without that privilege enters the site, they would be redirected to another site.

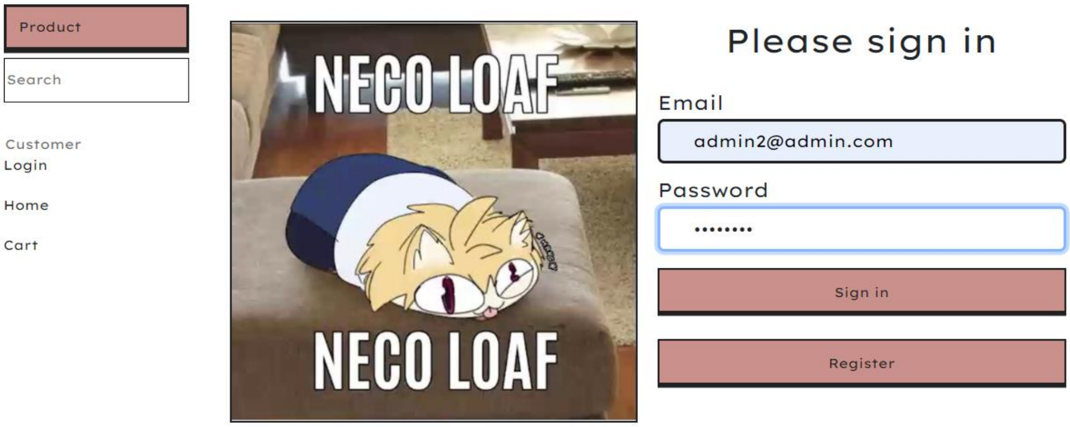


```
access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/mod, roles: ROLE_STAFF }
    - { path: ^/profile, roles: ROLE_USER }
    - { path: ^/cart, roles: ROLE_USER }
```

*Figure 41 access_control*

# 4. Screenshots

## 4.1. Log in



Please sign in

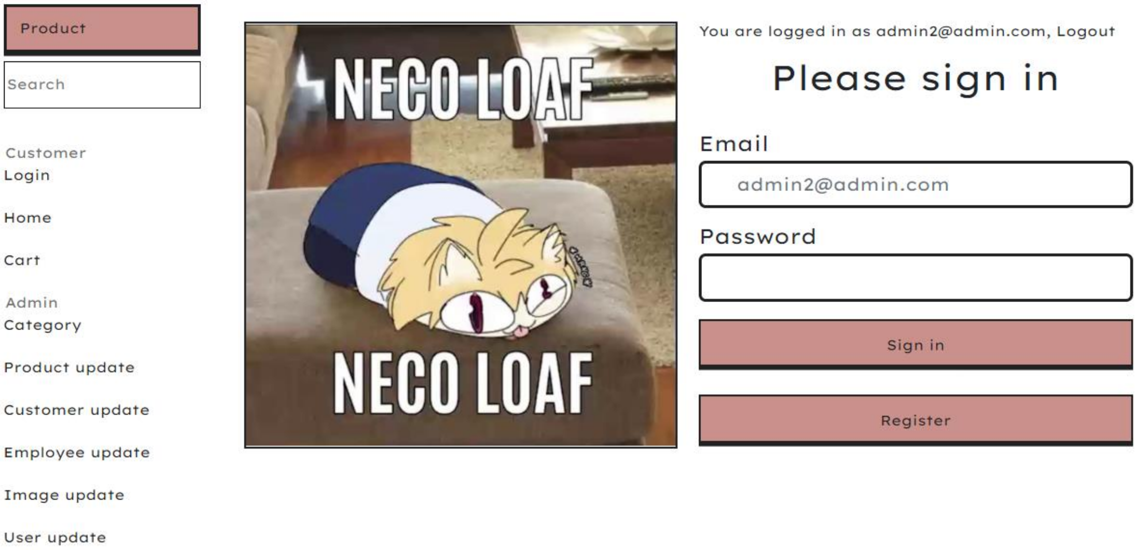
Email

admin2@admin.com

Password

••••••••

*Figure 42 Before logging in*



You are logged in as admin2@admin.com, Logout

Please sign in

Email

admin2@admin.com

Password

*Figure 43 After logging in*

## 4.2. Registration

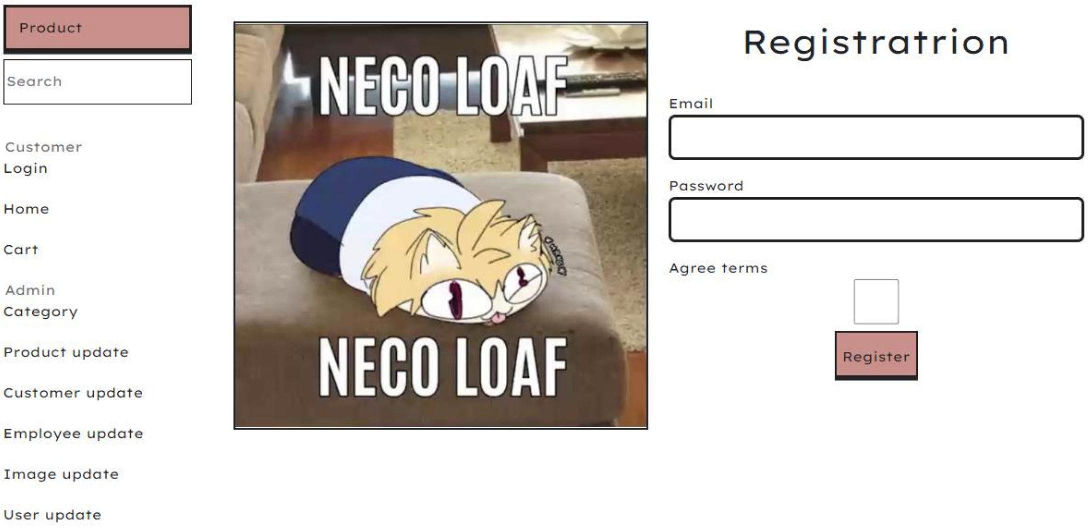


*Figure 44 Registration page*

## 4.3. Index

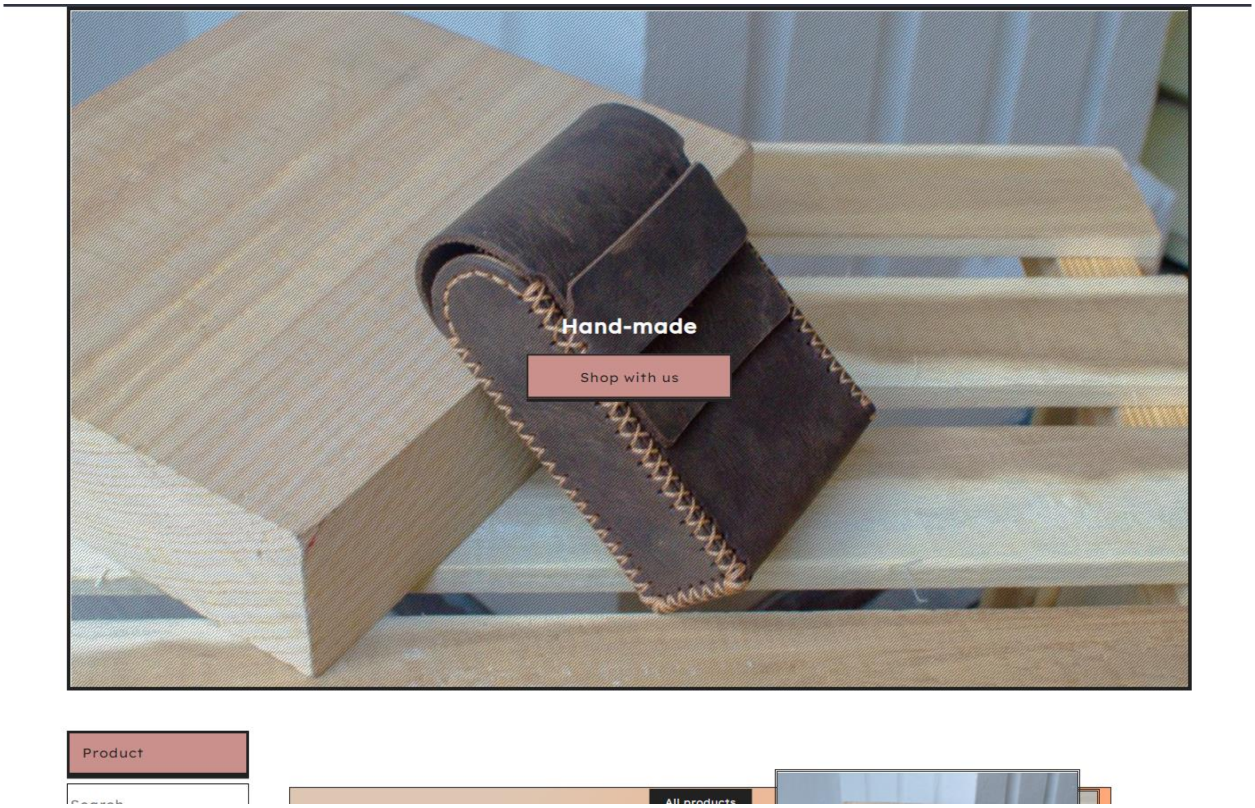


*Figure 45 Welcome message*

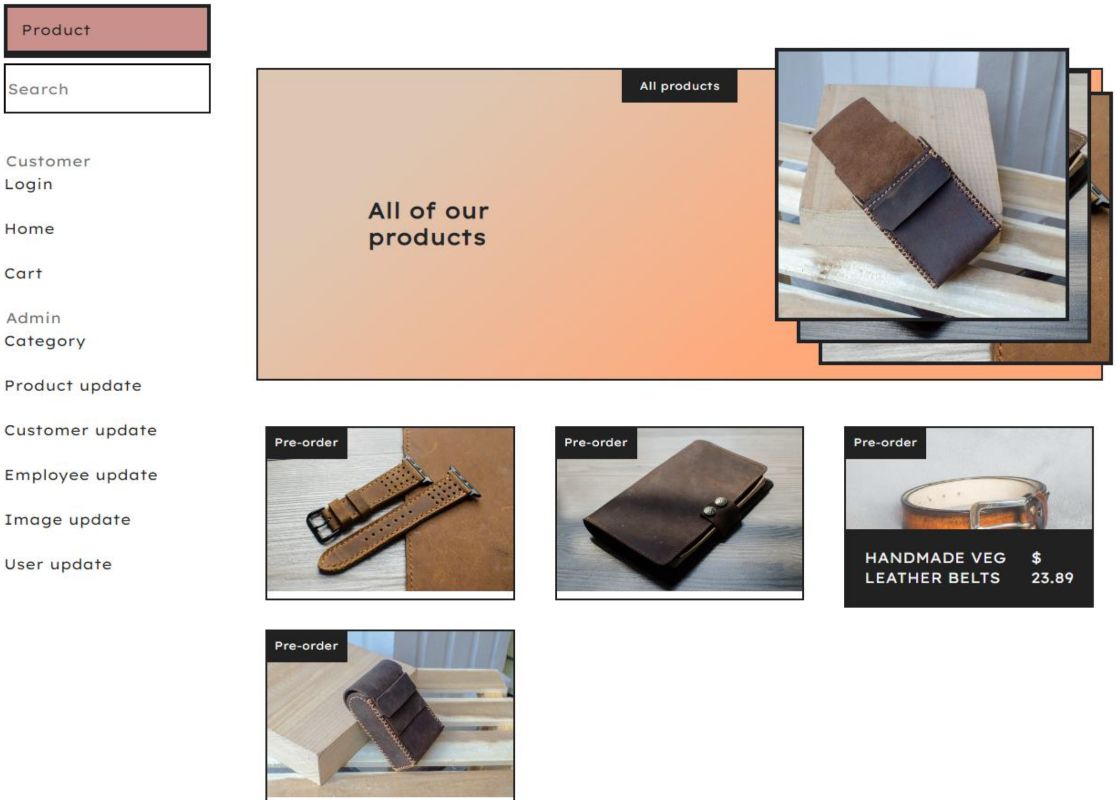


*Figure 46 Product list*

## 4.4. Product detail

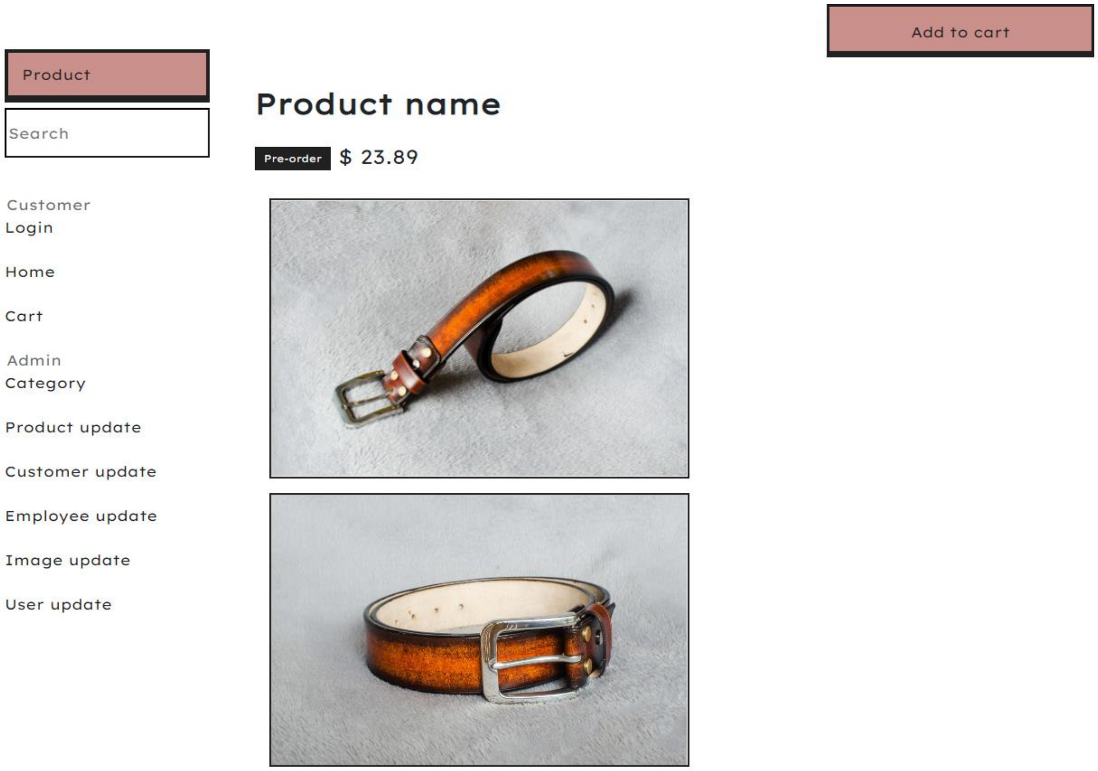
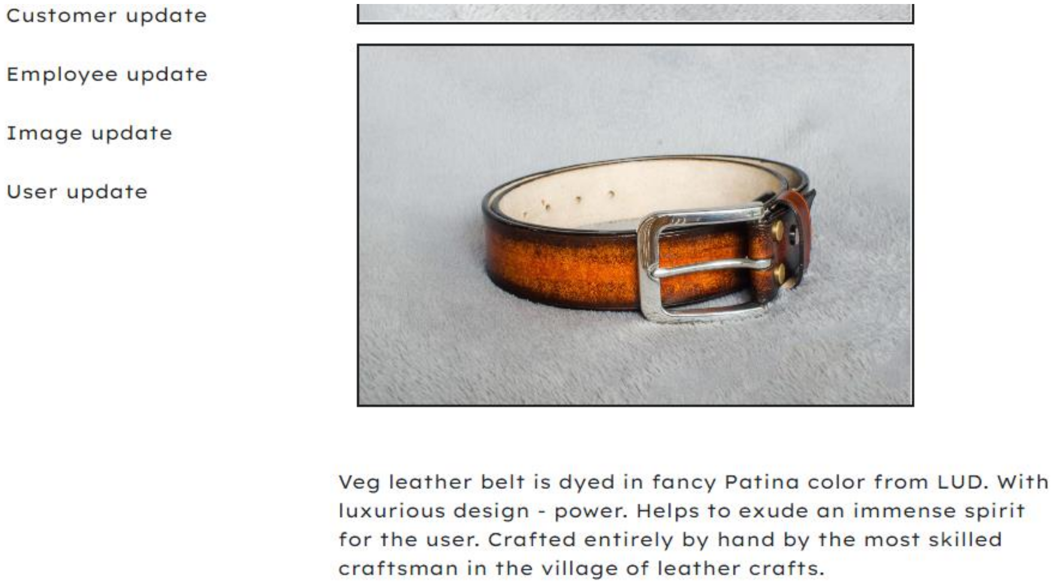


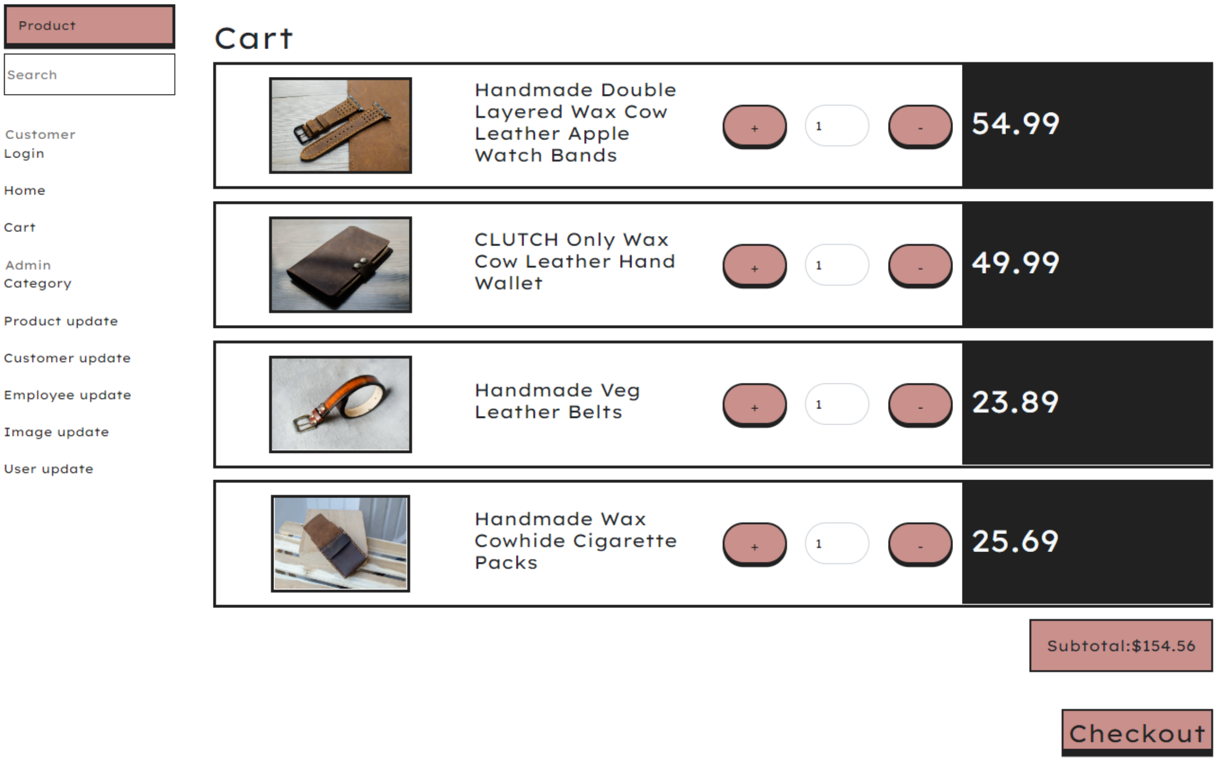
*Figure 47 Product detail*



Veg leather belt is dyed in fancy Patina color from LUD. With luxurious design - power. Helps to exude an immense spirit for the user. Crafted entirely by hand by the most skilled craftsman in the village of leather crafts.

*Figure 48 Product detail*

## 4.5. Cart



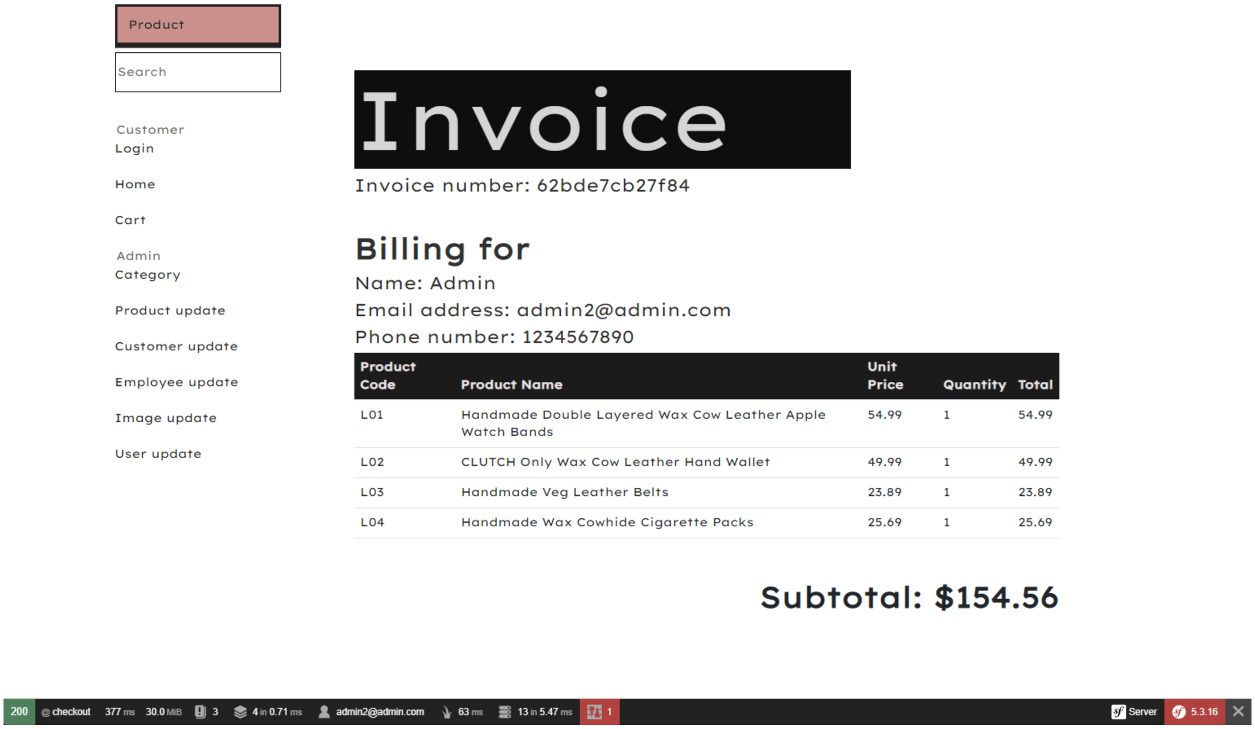
Cart

| | | | |
|---|---|---|---|
| Handmade Double Layered Wax Cow Leather Apple Watch Bands | + | 1 | - | 54.99 |
| CLUTCH Only Wax Cow Leather Hand Wallet | + | 1 | - | 49.99 |
| Handmade Veg Leather Belts | + | 1 | - | 23.89 |
| Handmade Wax Cowhide Cigarette Packs | + | 1 | - | 25.69 |

Subtotal:$154.56

*Figure 49 Cart*

### 4.6. Checkout



Invoice

Invoice number: 62bde7cb27f84

Billing for

Name: Admin

Email address: admin2@admin.com

Phone number: 1234567890

| Product Code | Product Name | Unit Price | Quantity | Total |
|---|---|---|---|---|
| L01 | Handmade Double Layered Wax Cow Leather Apple Watch Bands | 54.99 | 1 | 54.99 |
| L02 | CLUTCH Only Wax Cow Leather Hand Wallet | 49.99 | 1 | 49.99 |
| L03 | Handmade Veg Leather Belts | 23.89 | 1 | 23.89 |
| L04 | Handmade Wax Cowhide Cigarette Packs | 25.69 | 1 | 25.69 |

Subtotal: $154.56

*Figure 50 Checkout page*

# IV. Conclusion

## 1. Advantages of the website

- The UI of the store is clear and aesthetic, fitting the artistic theme of the products.
- The password is encrypted using hash, keeping them safe in case the system is breached.
- There is a search input for the user to search for the name of the product that they want.
- Product search function search for products by their likeness to the keyword instead of exact match, improving the user experience.
- When the user checkouts, an invoice with customer information is automatically generated along with the price.

## 2. Disadvantages of the website

- When the “Product” button is clicked, a list of categories is supposed to be displayed, however, the implementation of this feature fails to do so.
- When the quantity of a product changes, the quantity of every other product also changes.
- When the user is already logged in, the login form is not supposed to appear, but it does.

## 3. Lesson learnt

- Due to some incompatibility between Bootstrap and Twig, some features such as collapse and carousel do not work as intended, compromising some of the features initially designed.

- Due to the inexperience of the developers, the database generated by Symfony differs from the designs made by them, rendering functions like changing product quantity in the cart to work not as intended

## 4. Future improvement

- The Cart Entity and Controller can be fixed so that the quantity of each item in the cart can be changed.
- When a user is already logged in and goes to the log in page, there can be a log out button instead of a login form.
- When a user checkouts, there can be a form in which they can put in payment and shipping information before generating an invoice.

# Appendix

## 1. Group role

| Member | Role | Group dedication (1-10) |
|---|---|---|
| **Trinh Duc Anh (leader)** | Code implementation: interface (frontend), User function, Login page | 10 |
| **Dao Trong Nghia** | Code implementation: Admin functions, decentralization, product information search. | 10 |
| **Nguyen Thu Ha** | Execute the code: crud orderinfo, cart and checkout | 2 |

## 2. Github link

The source code to this project can be found at https://github.com/wildman9x/GCH1002-Web301-Assignment-ShoppingCart.

## References

Malewicz, M., 2022. *Neubrutalism is taking over the web.* [Online]
Available at: https://hype4.academy/articles/design/neubrutalism-is-taking-over-web
[Accessed 01 July 2022].