Bulgarian Diploma Thesis

Organic food e-commerce website

Radostina Kisleva, ID: 200035409

Student: [illegible]**, Date: 6 April 2021**
signature

Supervisor: John Galletly , Date: 6 April 2021
signature

Department of Computer Science, AUBG
Blagoevgrad, 2020

Title: Organic food e-commerce website

Author: Radostina Kisleva

Abstract:

The current thesis outlines the development of a web application project which is an eCommerce solution for an imaginary store for organic food. The website was developed using .NET for the backend and Angular for the frontend and offers various functionalities and features for looking at products and placing orders in the eCommerce website. The design of the website is user-friendly and engaging. The thesis starts with a detailed description of the application and the functional and non-functional requirements with a detailed analysis of the functional requirements through use case diagrams and use case narratives. After that, a description of the software architecture, organization of code and the design patterns which were used is provided. The security measures which have been implemented are described as well. The "Implementation" part deals with the methods through which the solution was implemented including describing the technologies used and explaining some of the code fragments. The acceptance tests which were performed are outlined in the Testing section, which also gives an account of the error-handling techniques. The last section details what was achieved and how the project can be extended in the future.

Declaration of authorship:

"The Senior Project/Bulgarian Diploma Thesis presented here is the work of the author solely, without any external help, under the supervision of John Galletly. All sources, used in development, are cited in the text and in the Reference section."

Author: Radostina Kisleva

[illegible]

Table of Contents

1. Introduction

1.1 Project Topic

It has been estimated that there are between 12 – 24 million eCommerce sites across the web with more and more created every single day. Having an eCommerce website is becoming a popular way to generate more profit and expand a business. The growth of eCommerce is astounding, and it is not slowing down anytime soon. In fact, it is becoming the preferred way in which customers shop which is convenient for buyers and sellers alike. That is why it is essential for businesses to have a strong online presence and present to their customers modern and engaging websites.

Ecommerce websites act as virtual stores in which customers walk down the digital isles of the business and fill up their carts with products. Then, they proceed to a secure payment portal and place their orders. Companies have to make sure they give high-quality eCommerce solutions to their customers and focus on making them engaging, with a high-quality user-friendly interface and various features which make the shopping experience effortless and smooth. Often the success of an eCommerce store depends not so much on the products being offered, but on the features and functionalities of the website.

1.2 Project Overview

The current project is an eCommerce solution for an imaginary store for organic food. The website includes all features which make a web store successful. Visitors are able to browse the inventory of products offered by the store. To make browsing easier, the website offers functionalities such as filtering the products by category and brand, sorting them by name and price and searching for an item by using a search term. The items are paginated and information about the current number of items which are being viewed is displayed on the page. Each item has a separate page which displays more detailed information about it, including a product description and several photos of the product. The customers have the option to add items to their virtual shopping carts, adjust the quantity of an item or remove it from their shopping cart. They are able to view the total price of all items in their carts and then proceed to a checkout page. The checkout page prompts the users to fill in their address and choose one of several shipping options. They are able to view their order summary and then proceed to payment. After completing all checkout steps and confirming the order, the customers see an order confirmation page along with a detailed order summary. The website will allow the customers to sign up and log in their personal user accounts through which they are able to access the items currently in their shopping carts, view their past orders and have their address stored for future purchases. In order to make the website easier to manage, there is a separate panel only for the administrators which allows them to add, edit or delete items in the store and add additional photos for each of the items.

1.3 Technologies used

The frontend part of the application is separated from the backend and different technologies are used. Backend is the portion of the website which is distant from the user's eye. It facilitates data management and interactions in an organized manner.

The technology which I have used to develop the backend is the ASP.NET Core framework. It is a free, open-source general purpose web framework developed by Microsoft. It provides features for building the backend of modern web applications, as well as web APIs. The programming language which is used for development with ASP.NET is C# which is a very powerful language that comes with a lot of handy tools.

Frontend of the website is the section which the user sees and interacts with. It is also referred to as client-side. To develop the frontend, I have used Angular. It is a TypeScript-based open-source web application framework led primarily by Google. TypeScript is the primary language for Angular application development. It is a superset of JavaScript with design-time support for type safety and tooling.

The rest of the frontend technologies are the usual protagonists - HTML and CSS. The Hypertext Markup Language (HTML) describes the structure of the page using markup while the Cascading Style Sheets (CSS) provide the visual and aural layout of the web pages. I have used Sass (Syntactically Awesome Style Sheets) which is a stylesheet language that is compiled to CSS. In other words, it is a CSS extension language that comes with some features that don't exist in CSS like variables, nesting, mixins, inheritance and more. Writing too much CSS can get tedious which is where Bootstrap comes to the rescue. Bootstrap is an open-source CSS framework. It contains CSS design templates for typography, forms, buttons, navigation and other interface components. I also used Font Awesome to get some of the icons for the website. The icons are created as scalable vectors, so they are high quality and work well on all screen sizes.

To manage the database and data storage, I have used SQLite which is a C-language self-contained full-featured SQL library. It is the most used database engine in the world. The advantages of using SQLite are its small footprint (it is very lightweight), self-containment (no external dependencies needed), user-friendliness (it is ready to use out of the box) and portability (an entire database can be stored in a single file).

To store the items in the user's shopping cart, I have used Redis. Redis is an open source, in-memory data structure store, which could be used as a database. Redis provides data structures such as strings, hashes, lists, sets and more. Its main purpose is to be used for caching. It is a very fast, persistent storage in memory. It can work as a key/value pair data structure with strings which is what I am going to use to store the basket's Id and the items in the basket.

For the integrated development environment, I have used Visual Studio Code – a free code editor made by Microsoft which is cross-platform. It offers many useful functionalities such as debugging, syntax highlighting, code completion, snippets, code refactoring and more to speed up the coding process. With the help of some extensions, I was able to use it for both the back and frontend parts of the application.

During the development, I have also used Postman which is an API client that makes the creating and testing of the APIs easy and a lot faster. I used it to create and save HTTP/s requests and read their responses in a convenient way rather than having to run the website in the browser every time. The result was a lot more efficient and less tedious work.

2. Specification of the software requirements and their analysis

2.1 List of functional requirements

The functional requirements describe what actions the developed software must support and what the application is supposed to accomplish for the user. They capture the intended behavior of the system. The following are the functional requirements which I have identified for the application. First are the functional requirements for the ordinary users that will purchase items in the store and then the ones for the administrator who will manage the store.

For the customers:

- Any customer should be able to register
- Any customer should be able to log in
- Any customer should be able to browse the catalogue of items
- Any customer should be able to view detailed information about an item
- Any customer should be able to filter the items by category or brand
- Any customer should be able to sort the items by name and by price (ascending and descending)
- Any customer should be able to search for an item
- Any customer should be able to add items to their shopping cart
- The information in the shopping cart should persist for further sessions of the user
- Any customer should be able to modify the contents of their shopping cart (change the quantity, add or remove items)
- Any customer should be able to see the total cost of their purchases in the shopping cart page
- When making an order, authenticated customers should be able to input their delivery address
- When making an order, authenticated customers should be able to choose one of the offered shipping options
- When making an order, authenticated customers should be able to see their order summary information
- When making an order, authenticated customers should be able to fill in their credit card information
- An authenticated customer should be able to place an order and see an order confirmation page with the order’s details
- An authenticated customer should be able to view details about their previous orders

For the administrator:

- Only authenticated administrators should be able to access the administrator panel
- An administrator should be able to create a new item
- An administrator should be able edit information about an existing item
- An administrator should be able to remove an item

- An administrator should be able to upload additional photos, set a main photo for an item and remove existing photos

2.2 Analysis of the functional requirements

I have divided the functional requirements into several use cases which include the main tasks the users will perform with the software. The use cases define a set of interactions between an actor and the system in order to achieve a goal. Below is the top-level use case diagram which includes the main use cases. They are View items, Manage Shopping cart, Make purchase, Register and Manage inventory. The "Manage shopping cart" use case can be used as a top-level use case or as a part of the "Make Purchase" use case. The Checkout use case is an included use case and it is not available by itself. It is only a part of making a purchase. The actors are web customers, which could be either registered or new (anonymous), and an administrator.

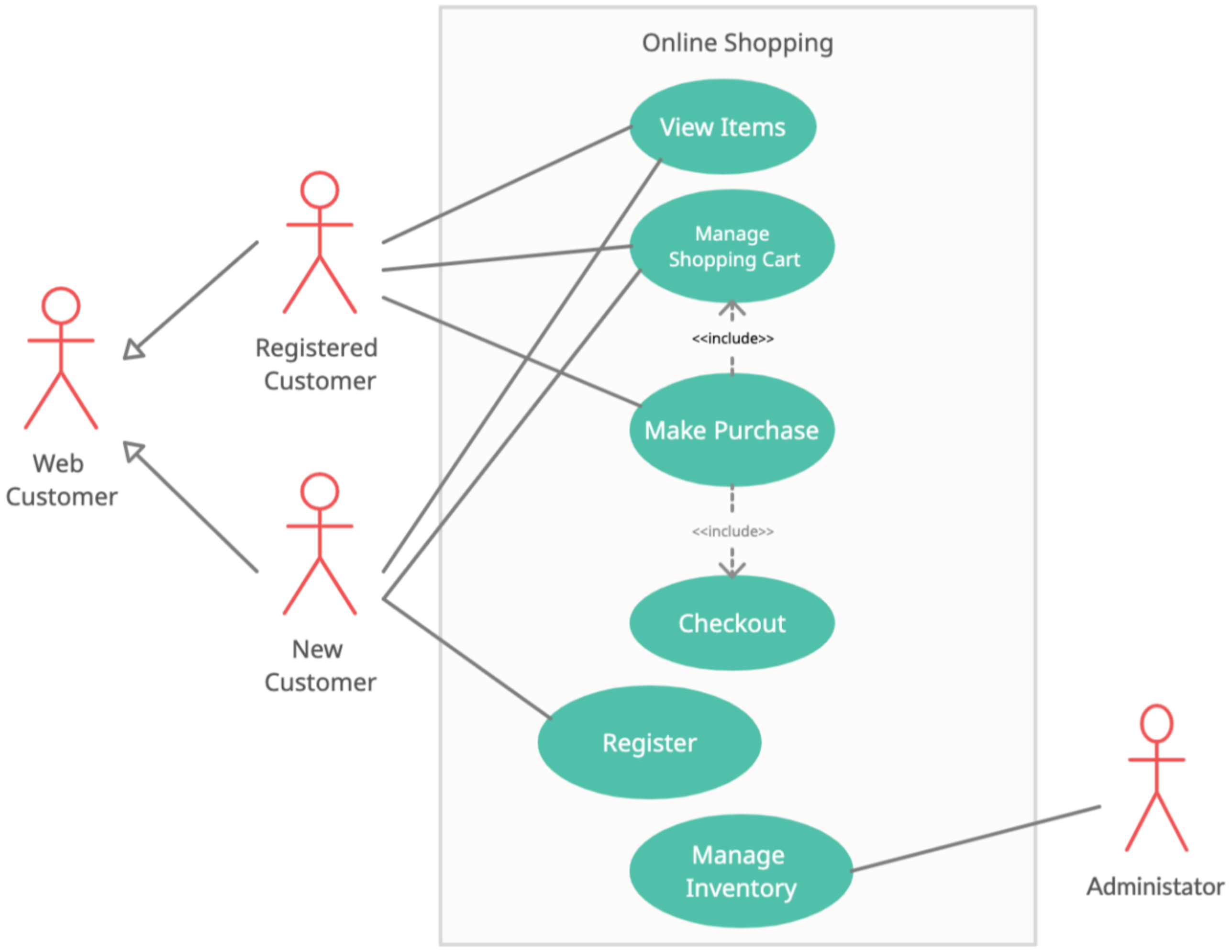


Figure 1 Top-level Use Case Diagram

In the following pages, I have provided a use case diagram for each of the main use cases. After that, I have given a detailed analysis and description of each functional requirement related to that use case. Each use case is initiated by a user which has a particular goal in mind and is considered successful if said goal is accomplished. Each use case narrative includes the goal in mind, the actor, any preconditions, triggering actions, the flow of events, exceptions that may occur, if any, and the post-condition.

<u>Use case #1: View Items</u>

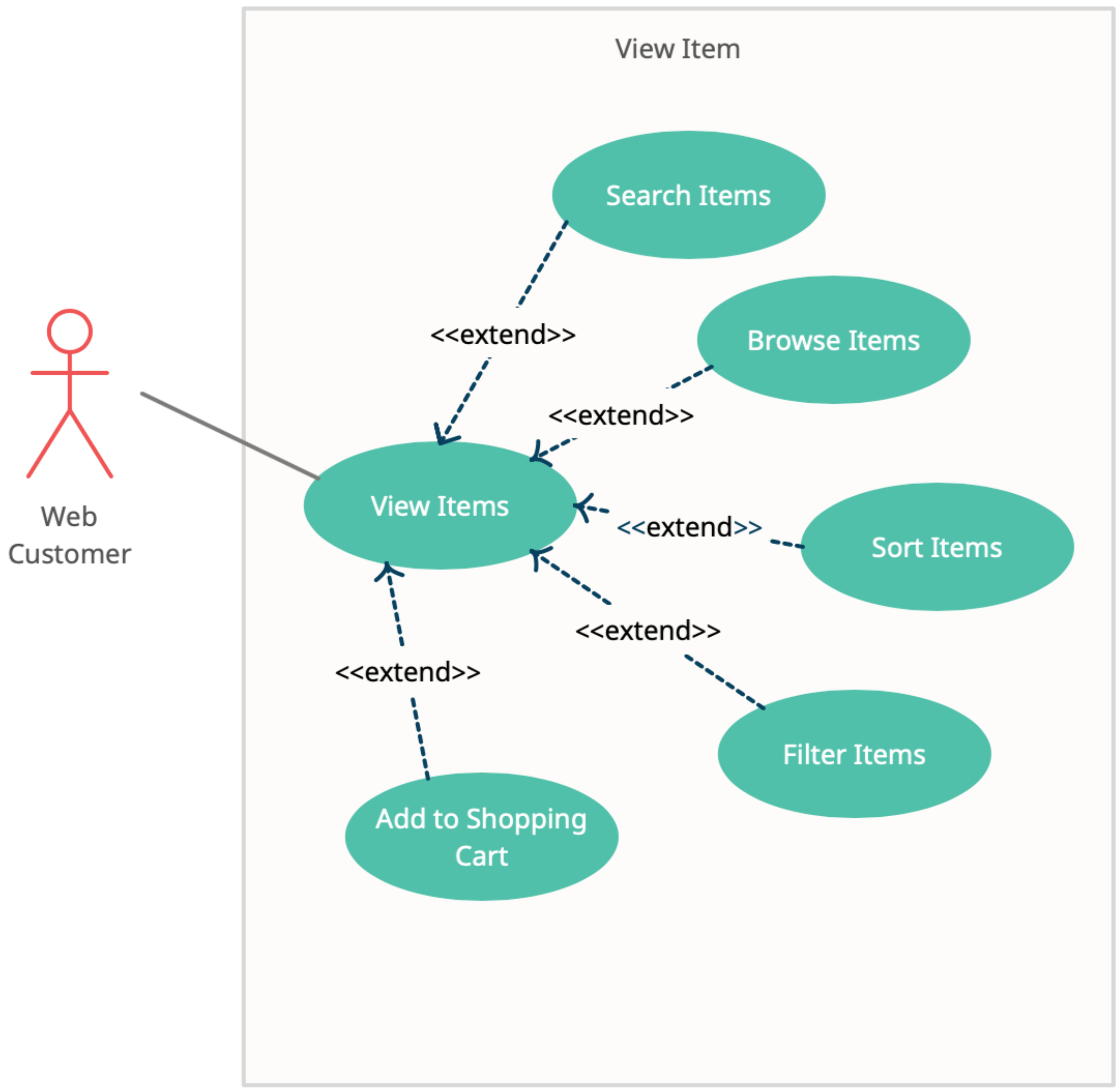


Figure 2 View Items Use Case Diagram

This use case consists of the customer browsing the website's inventory of products. The customer's goal is to look at the items and add some of them to their shopping cart. In order to help them browse the catalogue of items more easily, the system provides the searching, filtering, paging and sorting services. These are depicted as extending use cases on the diagram because they provide some optional functions which help the customer find an item. The customer does not need to be authenticated to browse the products and add a product to their shopping cart. The analysis of the functional requirements included in this top-level use case is given below.

- <u>Any customer should be able to browse the catalogue of items</u>

The web customers should be able to see the catalogue of items offered by the store. The provided user interface should be intuitive and easy to navigate. The user should be able to see the products arranged in a grid with the photo, name and price of each individual product displayed, along with two buttons – one which redirects them to the products details page and the other which adds the product to their basket.

Use case narrative:

The goal the customer has in mind in this use case is to add items to their shopping cart which later they might purchase. The actor in this use case is the user. There are no preconditions except that the user should have navigated to the website's URL. Authentication is not required here, and any user should be able to get to this page. The use case starts when the user clicks on the "Shop" option from the navigation bar. The typical flow of events is the following:

1. User clicks on "Shop" link from the navigation menu
2. System displays all of the items available in the store sorted alphabetically by default, with 6 items appearing per page with the additional dropdowns and tables used for filtering and sorting
3. User clicks on the different pages and looks at the items available.

No errors and exceptions should occur at this stage. The postcondition is that the items available in the store are displayed and the user can look at them.

- Any customer should be able to view detailed information about an item

When the customer clicks on the product's name or on the designated "Details" button available for each one of the items, they should be taken to a page where more information about the particular item will be displayed such as its picture on a larger scale, its description and price. Here the users can also see additional pictures of the particular product which are displayed in a mini gallery under the main photo. From this page, the customers should be able to add the product to their cart and adjust the quantity before that, if needed.

Use case narrative:

The goal of this use case is to allow the customer to see more information about a particular item, see more images or add more than one quantity to their shopping cart. The actor in this use case is the user. The precondition is that the user should be located on any of the pages on which the items' names can be clicked. Authentication is not required. The use case starts when the user clicks on the name of any of the items or on the designated "Details" button on the "Shop" page for any of the items. The typical flow of events is the following:

1. User clicks on "Details" button next to an item (or item's name)
2. System displays the page with details for the particular item
3. User looks at the additional information about the item
4. (Optional) User can click on any of the images in the gallery
5. (Optional) User can increase or decrease the quantity of the item

The postcondition of the use case is that the item's "Details" page has been displayed and the user has seen it. No errors and exceptions should occur at this stage.

- Any customer should be able to filter the items by category or brand

The “Shop” page should contain two tables – one with all brands available in the store and the other with all categories. By clicking on the name of a particular brand or category from these tables, the system should respond accordingly by displaying items only from that brand, category or both if two filters are chosen simultaneously. If the user wants to remove the filters, they should click on the “All” option in the tables.

Use case narrative:

The goal of this use case is to allow the customer to see only the products from a particular brand or category. The actor is the user. The precondition is that the user is located on the website’s Shop page because that’s where the tables with the brands and categories are located, as well as the view with all of the products. The use case starts when the user clicks on any one of the rows with a category or brand in the tables. The postcondition is that only the products from that particular brand or/and category are shown. No errors and exceptions should occur at this stage.

- Any customer should be able to sort the items by name and by price (ascending and descending)

The “Sort” dropdown menu offers the users three options to choose from: Sort Alphabetically (default option), Sort by ascending price and Sort by descending price.

Use case narrative:

The goal of the use case is to display to the customer the items sorted by name or ascending price or descending price, based on their preference. The only precondition is that the customer is located on the “Shop” page because that is where the Sort dropdown menu and the view with all of the products are located. The use case starts when the customer clicks on any of the options in the “Sort” dropdown menu. The system returns the items sorted by the chosen option. If a filter for a category or brand has been applied before the sorting, only the filtered items are sorted and shown. The postcondition is that the items are now sorted by a certain criterion. No errors and exceptions should occur at this stage.

- A customer should be able to search for an item

By typing a search term in the search box above the items, users should be able to see all of the items which include the search term in their name.

Use case narrative:

The goal the customer has in mind is to find a particular item faster or find all items which contain a particular word in their name instead of browsing the whole catalogue and looking for it manually. The precondition for this use case is that the user is located on the website’s “Shop” page. They should have also typed the word by which they want to search in the designated search box. The use case starts when the user clicks on the “Search” button. The system returns only the items that contain the search team in their name or nothing if such an item does not exist.

<u>Use case #2: Manage shopping cart</u>

After they have browsed through the items, the users can choose to add some of them to their virtual shopping carts. Besides adding items to the cart, after they have done that, the users should have the ability to modify the contents of their carts which means increasing or decreasing the quantity or removing an item from the cart if they have changed their mind about buying it. This use case also includes a service which calculates the price for each particular item based on the items' quantity and price. After that, all of those are summed in order to calculate the total for the whole basket. First, the use case diagram for this use case is given below and then the functional requirements related to it are analyzed one by one by providing a detailed use case narrative.

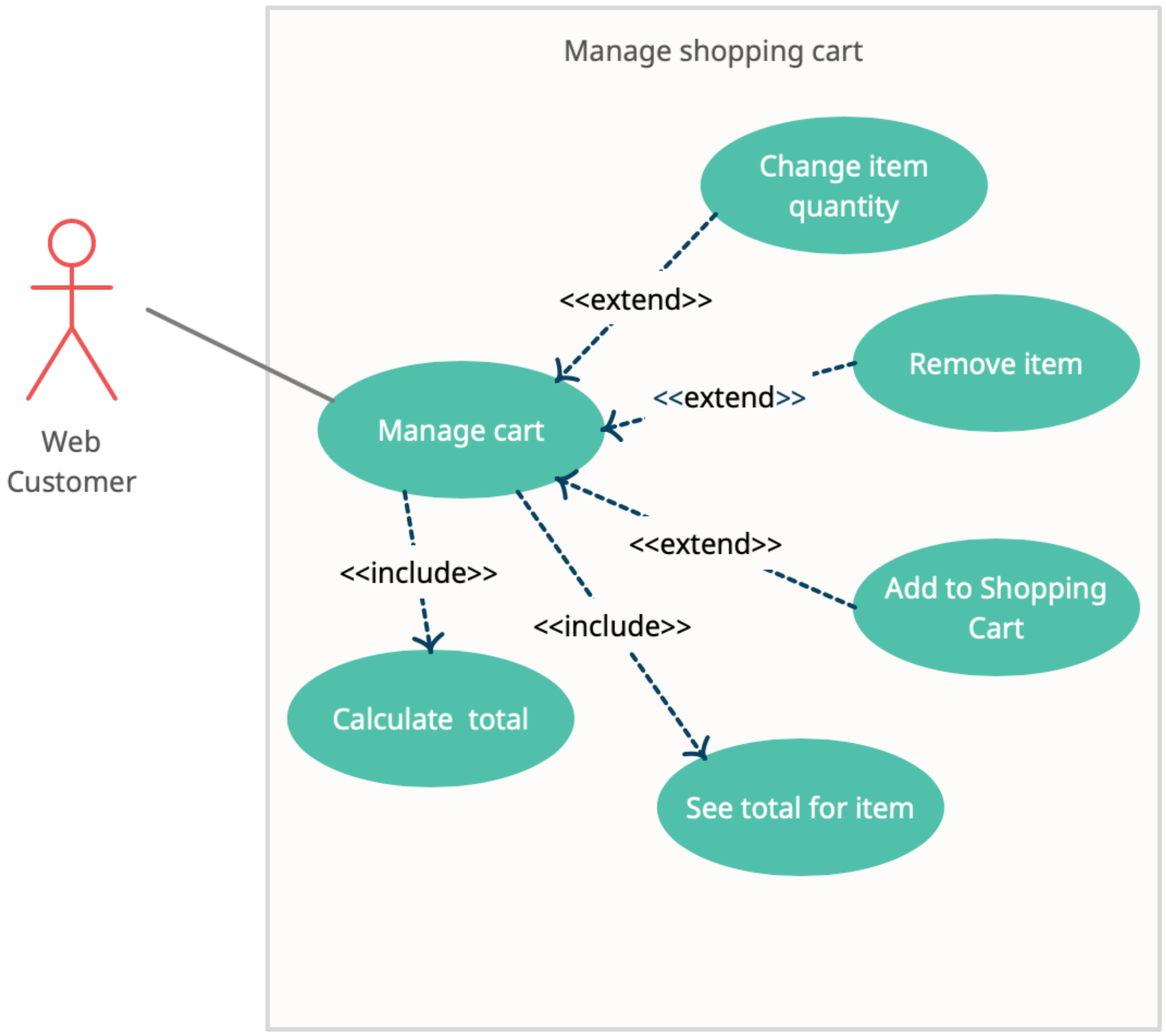


Figure 3 Manage Shopping Cart Use Case Diagram

- <u>A customer should be able to add items to their shopping cart</u>

<u>Use case narrative:</u>

Without a shopping cart functionality, eCommerce sites are not that anymore. The goal of the use case is to allow the customer to save a particular item and potentially purchase it at a later point in time. The only actor in this use case is the user. The precondition is that the user is located on the "Shop" page or on an item's "Details" page because on those pages the "Add to Cart" buttons can be found. Authentication is not required here, and any user can create a shopping cart regardless they are logged in or not. The use case starts when the user clicks on the button with a shopping cart icon for an item. The typical flow of events is the following:

1. User clicks on shopping cart button for a particular product
2. System adds the product to the user's shopping cart
3. Shopping cart items counter in the navigation bar increases
4. User is able to see the item's name, photo, quantity and price in their shopping cart page

The postcondition is that now there is a new item added to the user's shopping cart. No errors and exceptions should occur at this point.

- <u>The information in the shopping cart should persist for further sessions of the particular user</u>

Normally users visit a website several times before they actually make a purchase. For that reason, the contents of the shopping cart should persist in local storage so that the users don't have to add the items each time they visit the web page. Having an account is not necessary in order to use this feature but the website should be visited from the same browser through which the basket was initially created.

- <u>A customer should be able to modify the contents of their shopping cart (increase or decrease the quantity, add or remove items)</u>

<u>Use case narrative:</u>

Users should have the ability to modify the contents of their shopping carts because making various changes before actually making the purchase is very common. The goal of this use case is to allow the customers purchase more than one quantity of a particular item and then decreasing that quantity if needed. In addition, they should be able to remove an item from the shopping car. The precondition is that the user has already added at least one item in their cart. Authentication is not required for changing the shopping cart. The use case starts when the customer clicks on any of the buttons available for a product in the table with all products. Here is the typical flow of events:

1. User clicks on shopping cart icon and sees the contents of their shopping cart.
2. User adjusts quantity by pressing the "+" or "-" buttons next to the quantity number for an item
3. System updates the quantity and adjusts the total price for the particular item
4. System updates the total price of all products accordingly

5. User clicks on the trash icon to remove a product from the shopping carts
6. Item disappears from the shopping cart
7. Shopping cart items counter in the navigation bar goes down
8. The total price of all items is adjusted accordingly

The postcondition is that the items in the shopping cart have changed as well as the total price for the order.

- <u>Any customer should be able to see the total cost of their purchases in the shopping cart page</u>

At the bottom of the page, below all of the items in the cart, users should see an "Order summary" table which includes the total cost of all items which are currently in the shopping cart. This number is updated automatically each time a user adds a new item, removes and items or changes the quantity of an item.

<u>Use case #3: Make purchase</u>

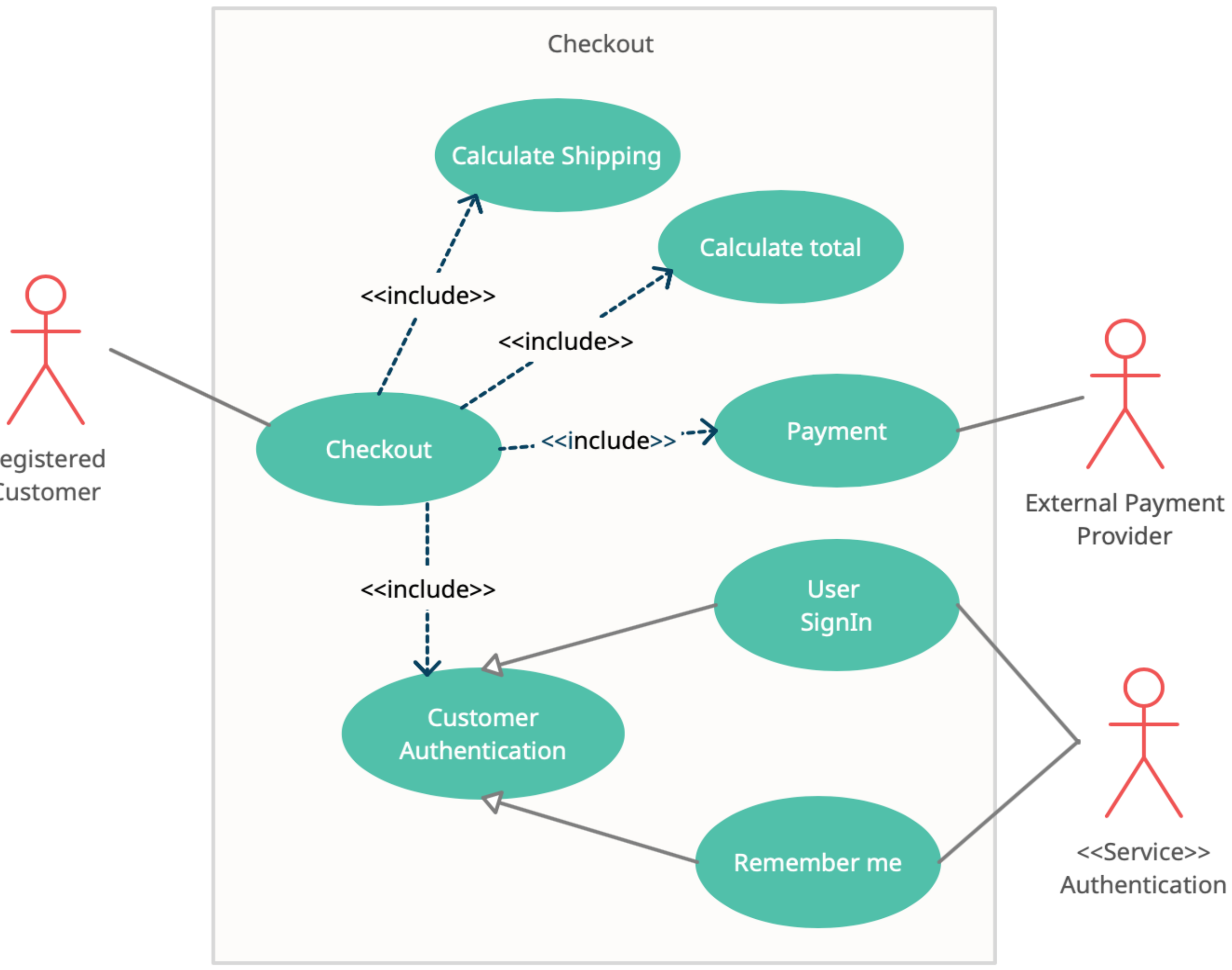


Figure 4 Make Purchase Use Case Diagram

After having selected the items to purchase from the store, the users can proceed to the checkout page. Only logged in users should be able to get to this page. Here they fill in their delivery address or the system populates it automatically if they have already made a purchase or have saved a default address. They also choose one of the four shipping options available and have its price added to the total price. The users are also able to review the order, input their credit card information and submit the order. After that they see a confirmation page which gives them the ability to view the order details. What follows is the analysis of the related functional requirements.

- <u>When making an order, authenticated customers should be able to input their delivery address</u>

Filling in all of the requested shipping address information is necessary when placing an order. The user will not be allowed to proceed to the next step of the checking out process if they have not filled all of the fields in this form. If the user has an address already saved in the system, the fields will be automatically populated with that address.

<u>Use case narrative:</u>

The goal of the use case is to get the customer's address so that the store knows where to deliver the items in the order. The actor is the user. The precondition is that the user has registered and is logged into their account. The claims in their token are checked before they are given access to this page. Authentication is required here. The other precondition is that the user has added at least one item to their shopping cart with a quantity of at least one. The user should also be located on the shopping cart page because this is the only place from where the Checkout page can be accessed. The use case starts when the user clicks on the "Proceed to checkout" button on the shopping cart page. The typical flow of events is the following:

1. System checks if customer is authenticated (if there is a valid token)
2. (Optional) System automatically fills the address form if an address has already been saved in the system
3. User inputs their first and last name in the designated fields
4. User inputs their street and city in the designated fields
5. User inputs their country and postcode in the designated fields
6. System performs checks after each field is input
7. (Alternative) System shows an error message in case any of the fields is left empty and does not the user proceed with next steps if this is the case
8. (Optional) User saves the address they have just input as a default address for future purchases
9. System saves the users address and allows them to go to the next stage

Since here there is user input involved, errors can occur. This will be the case if the user leaves some of the fields of the form empty. Error messages should appear to notify them of what is wrong. The postcondition is that the system has saved the user's address and they are now allowed to go onto the next step of the checkout process. If the user has chosen to save the current address as default, the next time they are checking out, the form fields will be automatically filled.

- When making an order, authenticated customers should be able to choose one of the several offered shipping options

The next step of the checkout process is to choose a shipping option. The users cannot proceed until they complete this step, and it is necessary in the checkout process. There are four shipping options available: overnight shipping, express ground shipping, regular group shipping and free shipping. They all have a different price depending on the time of delivery.

Use case narrative:

The goal of this use case is to get the customer's preferred shipping method so that they can be charged accordingly and so that the store knows how to send the order. The actor is the user. The precondition is that the user has completed the address form from the previous use case correctly. The use case starts when the user clicks on the "Go to Delivery" button at the bottom of the address checkout form. The typical flow of events is the following:

1. User chooses one of the shipping options by clicking on the radio button next to its name
2. System updates the total cost based on the price of the chosen option
3. Shipping option is saved and added to the order

No errors and exceptions can occur at this point. The postcondition is that the user can now proceed to next step of the Checkout process and has the shipping price added to their total and the preferred shipping method saved to the order.

- When making an order, authenticated customers should be able to see their order summary information

The third step of the checkout process is the review step in which the user sees all of the items which are a part of the current order with their image, name, individual price, quantity and total price.

Use case narrative:

The goal of this user case is to display to the user one last time all of the items which will be a part of their final order. This is the last confirmation needed from them before moving on to paying and placing the order. The actor is the user. The precondition is that the user has completed the previous step and has chosen a shipping method. The use case starts when the user clicks on the "Go to Review" button in the shipping options form. No errors can occur at this stage since no user input is required. The postcondition is that the user can now proceed to the next step of the checkout process and that they have reviewed and confirmed the items in their order.

- When making an order, authenticated customers should be able to fill in their credit card information

The last step of the checkout process is the Payment step. Here the users can input their credit card information. The website is currently not connected to an online payment processing platform which will accept actual credit cards and verify them. Nevertheless, if there such a service was implemented, the customers would have to input their credit card information at this stage.

Use case narrative:

The goal of this use case to get the customer's credit card information so that they can be charged for their purchase. The actor is currently only the user but, in the future, an online payment platform will be the second actor. The precondition is that the user has reviewed the items in the previous checkout step. The use case starts when the user clicks on the "Go to Payment" button at the bottom of the review component. The typical flow of events is the following:

1. User inputs the name on card, the card number, expiration date and the CVC code.
2. System verifies if all the fields have been filled
3. System returns error if one of the fields is empty
4. (Future work) Third party online payment platform verifies the credit card information and charges the card.

The postcondition is that the customer has now paid for their order and can now confirm it and place it officially.

- An authenticated customer should be to place an order and see an order confirmation page with the order's details

Use case narrative:

The goal of this use case is to get a last confirmation from the user about placing their order and then saving the order in the database. The store will be officially notified of the user's order and can start preparing it. The actor in this use case is the user. The precondition is that the user has completed the four previous steps in the checkout process, the last of which was the payment. The use case starts when the user clicks on the "Submit Order" button at the bottom of the payment form. The typical flow of events is given below. No errors and exceptions can occur at this stage because all that is required is the click of a button.

1. User is ready with the checkout and clicks "Submit order" button.
2. Order is saved in the database
3. User's basket is automatically deleted
4. System redirects the user to a page which says that the order is confirmed
5. (Optional) User chooses to view their order.
6. (Optional) System redirects user to a page where they can see a table with the products ordered, their quantity and price as well as the total price of the whole order including shipping.

The postcondition is that the user's order has officially been recorded in the database. The user is notified that they have placed the order successfully. Optionally they have viewed the items which will be a part of their order.

<u>Use case #4: Manage inventory</u>

The administrator of the website has the ability to see all of the products offered by the website, add a new product, edit details about existing products and remove a product Moreover, administrators have the ability to add more images for the products, delete images and change the main photo for the item which is displayed in the "Shop" page. The administrative interface is separate from the common functions performed by users and can only be accessed if the user is in the "Admin" role.

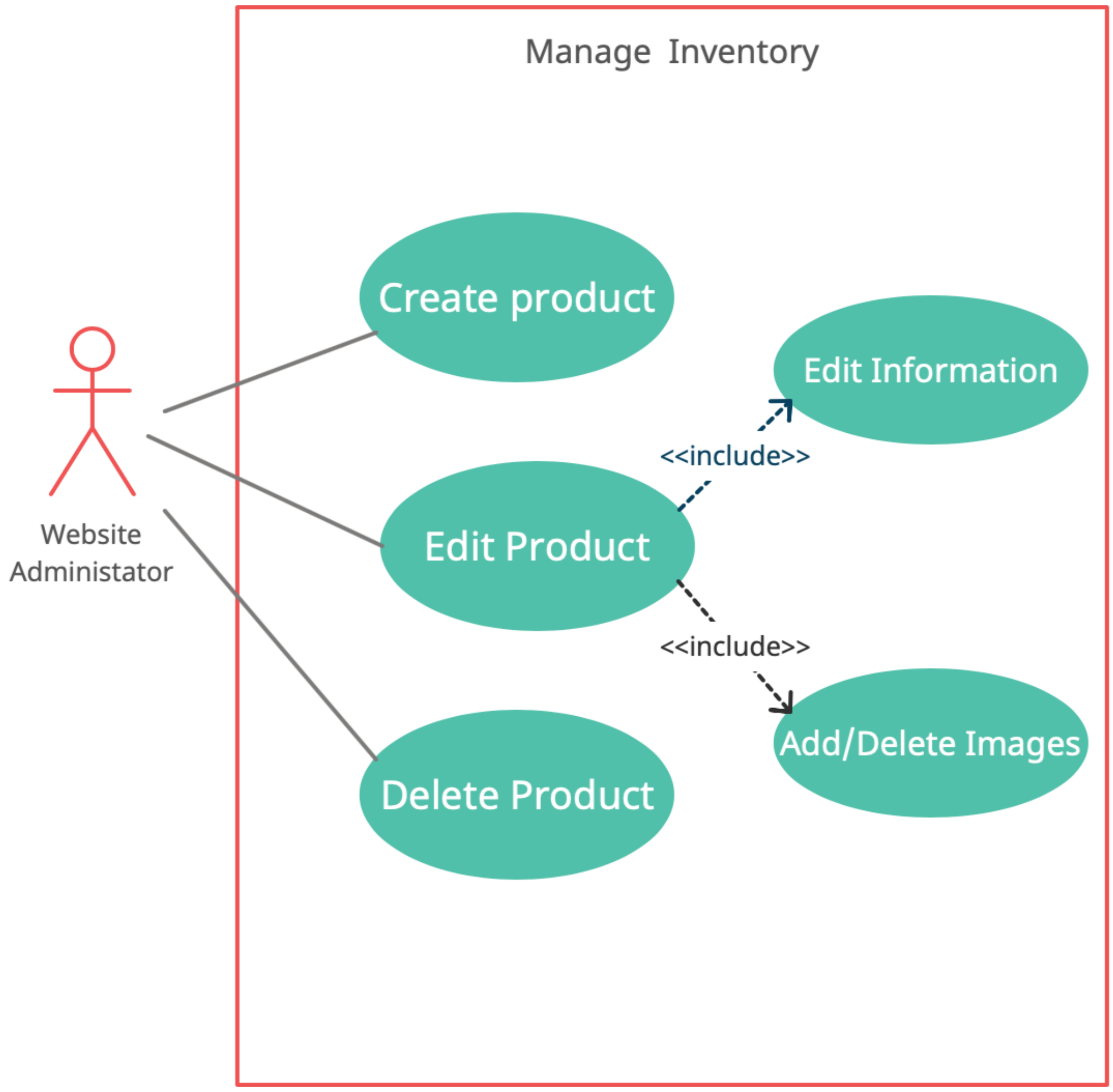


Figure 5 Manage Inventory Use Case Diagram

- <u>Only authenticated administrators should be able to access the administrator panel</u>

When any user logs into the system, the claims from their token are decrypted and checked. The token includes the claims that the user has and one of the claims is the role. If the user's role is set to "Admin", they are allowed to access the admin panel and make changes in the inventory of the products. Validation is performed both on the client and on the server.

- An administrator should be able to create a new item

Use case narrative:

The goal of this use case is to allow the administrator to add a new item to the inventory of products which the customers will be able to view and add to their shopping carts. The actor in this use case is the administrator. The condition is that the user performing this task is in an administrator role. The claims from their token should be checked and the token should be verified before they are allowed to get to the admin panel. The admin should be located on the “Admin” page on the website. The use case starts when they click on the “Create new product” button at the top of the table. The flow of events is the following:

1. Admin clicks “Create new product” button
2. System redirects them to the item creation form
3. Admin fills the three required text fields (name, price, description)
4. Admin chooses values in the two required dropdown menus (brand and category)
5. System verifies the information after each text field has been filled
6. (Alternative) System displays error messages if any of the fields is left empty
7. (Alternative) System displays error message is the price is left as 0
8. Admin clicks “Submit” button
9. New product is created and added to the database

Errors can occur here because user input is involved. The conditions are that the user inputs all of the fields and that the price is no less than 0. The postcondition is that a new item has now been added to the database and it can be viewed on the “Shop” page with the rest of the products. The users can now look at it and buy it.

- An administrator should be able to edit information about an item

Use case narrative:

The goal of this use case is to allow the administrator to edit an item. Very often in an online store prices change when there are promotions or products’ images need to be updated with newer ones or there might be outdated and wrong information about some of the items. The actor in this use case is the administrator. The condition is that the user performing this task is in an administrator role. The claims from their token should be checked and the token should be verified. The admin should be located on the “Admin” page on the website. The use case starts when they click on the “Edit” button in the table row of the item which they would like to edit. The flow of events is the following:

1. Admin clicks “Edit” button for a product
2. System redirects admin to the item’s “Edit form”
3. The input fields are already populated with the item’s information.
4. Admin makes changes
5. (Alternative) System returns error messages if any of the fields is left empty.
6. (Alternative) System returns an error message if the price is 0
7. Admin clicks “Submit” button.

The postcondition is that the product's information is now updated, and the changes are visible on all of the views which contain the product's information.

- An administrator should be able to upload additional photos, set a main photo for an item and remove existing photos

Use case narrative:

The goal of this use case is to have the image/s of a particular item altered in some way whether it will be adding new ones, deleting some of them or changing the main photo for an item. The actor in this use case is the administrator. The condition is that the user performing this task is in an administrator role. They should have chosen an item to edit by clicking on the "Edit item" button. The system should have redirected them to the edit item form. The trigger for this use case is when the administrator clicks on the "Edit photos" option from the item edit form. I have separated the flow of events into several parts depending on whether the goal is to add or remove an image or set a new main photo.

Add a new image flow of events:

1. Admin clicks "Edit photos" button
2. System shows all images available for the item
3. Admin clicks "Add new photo" button on the top of the gallery
4. Admin drops an image in the placeholder
5. Image is uploaded and the admin can see it in the preview
6. (Alternative) Image is not uploaded if its format doesn't correspond to an image
7. (Optional) Admin can resize(crop) the image
8. (Optional) The preview of the image changes in accordance with the resizing (cropping)
9. Admin clicks "Upload image" button
10. (Alternative) Admin clicks "Cancel" button and the process is terminated

The exception that could occur in this use case is that the admin might choose to upload a file which does not have the .jpg or .png file format. If this happens, the image is not uploaded. The postcondition for this use case is that now there are additional photos for the image which the user can view in the product's details page.

Removing an image flow of events:

1. Admin clicks "Edit photos" button
2. System shows all images available for the item along with two buttons
3. Admin clicks the trash can icon button below a photo to remove it
4. Photo is erased from the gallery
5. (Alternative) Photo is not erased if this is the main photo of an item

The exception that could occur in this use case is the admin trying to erase the main photo of a product. This is not allowed. First, a new main photo should be chosen and then the previous main photo can be deleted. The postcondition is that now the image has disappeared from the item's image gallery.

<u>Setting a new main photo flow of events:</u>

1. Admin clicks “Edit photos” button
2. System shows all images available for the item
3. Admin clicks “Set main” button for a photo
4. The main photo for the item is updated in all pages where it is needed

The item now has a new main image in the “Shop” page, basket, details pages and admin table with all products.

- <u>An administrator should be able to remove an item</u>

<u>Use case narrative:</u>

The goal of this use case is to allow the administrator to remove an existing item from the inventory of products which the customers will no longer be able to view and add to their shopping carts. The actor in this use case is the administrator. The condition is that the user performing this task is in an administrator role. The admin should be located on the “Admin” page on the website. The use case starts when they click on the “Remove” button in the table row of the item which they would like to remove. No exceptions or errors should occur. The postcondition is that the item can no longer be viewed on the “Shop” page and it is removed from any user’s baskets.

<u>Use case #5: Register/Login</u>

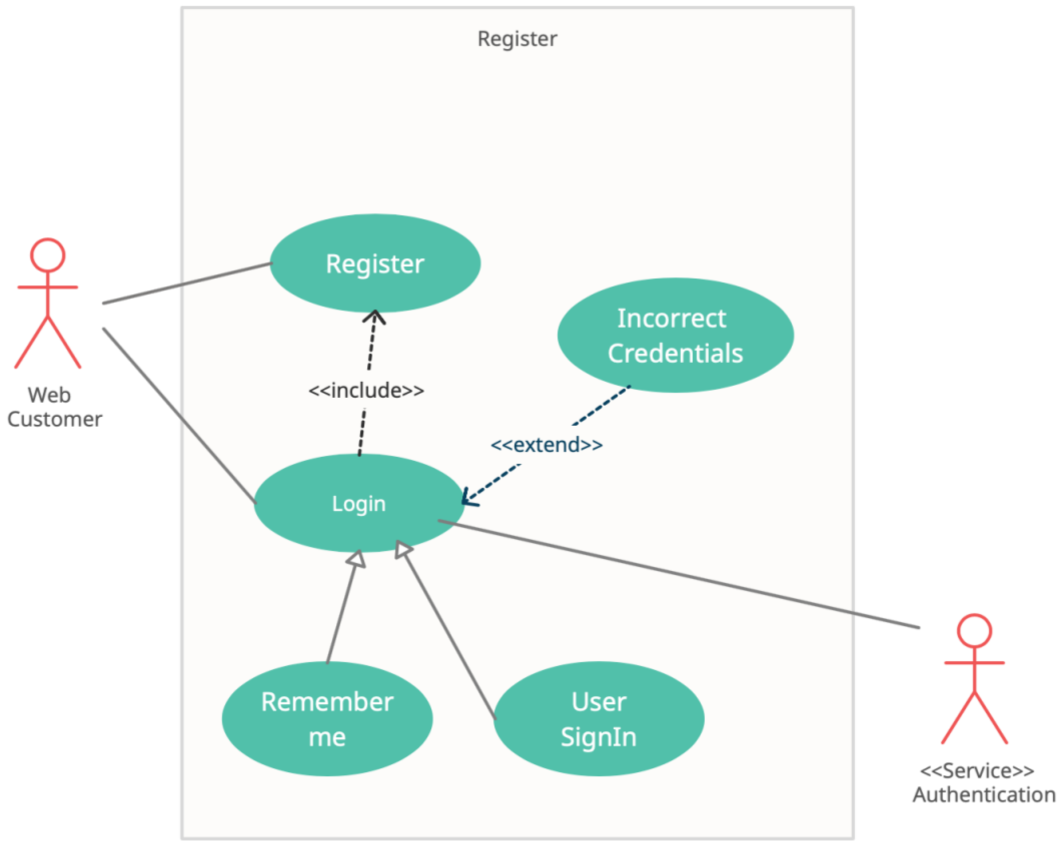


Figure 6 Register Use Case Diagram

This use case includes the registration and login functionality of the system. Users should be able to make an account so that they can place orders. Moreover, by having an account, they will be able to save a default address and see their previous order history. The user's login is persisted into their browser by default so that they don't have to login every time they visit the page. They can log out by using the "Log out" button in the navigation bar. The functional requirements related to this use case are the following:

- Any customer should be able to register

Use case narrative:

The goal of this use case is to allow customers to create an account in the application. This is needed in order to place an order. The store needs to know their name and email in case they need to contact them with any enquires about their order. The actor in this use case is the user. The condition is that the user is located on the website's URL. It doesn't matter which page they are on because the "Sign up" button is always available in the navigation bar. The use case starts when the user clicks on the "Sign up" button in the navigation bar. The flow of events is the following:

1. User is redirected to the "Register" page
2. User fills in the required name field
3. System validates the input and accepts it if it is valid
4. (Alternative) System returns an error message in case the field is left empty
5. User fills in the required email field
6. System validates the input and accepts it if it is valid
7. (Alternative) System returns error messages in case the field is left empty, or the input does not have the structure of an email or the email has already been used for registering
8. User fills in the required password field
9. System validates the input and accepts it if it is valid
10. (Alternative) System returns an error message in case the field is not completed, or the password is not complicated enough (too short, any of the required characters is missing)
11. User creates an account in the system

Quite a lot of errors with the user input can occur here which should be handled properly by the system. If the user corrects all of them, the postcondition is that now they will be able to log in the system with their credentials which in turns means they will be allowed to place orders. Users should be automatically logged into their account after successful registration.

- Any customer should be able to log in

Use case narrative:

The goal of this use case is to allow customers to log in the account which they have created. The actor in this use case is the user and an Authentication service. The precondition is that the user is located on the website's URL. It doesn't matter which page they are on because the "Login" button is always available in the navigation bar.

The use case starts when the user clicks on the “Login” button in the navigation bar. The flow of events is the following:

1. User is located on the “Login” page
2. User fills in the required email field
3. System validates the input
4. (Alternative) System returns error messages in case the field is left empty, or the input does not have the structure of an email
5. User fills in the required password field
6. System validates the input
7. (Alternative) System returns an error message in case the field is left empty
8. User presses the “Login” button
9. User is logged in their personal account
10. A token is generated for the user which includes their claims
11. (Alternative) System returns error messages in case the email and password commination has not been registered in the database

Some errors with the user input can occur here which should be handled properly by the system. If the user corrects all of them, the postcondition is that now they will be logged in their account which means that they can place orders or view their past orders. If not, users will be denied access to their accounts until they resolve the issues.

- An authenticated customer should be able to view details about their previous orders

Users should have the ability to see all of the orders made from their account, along with the order’s total price, status and date. Additionally, by clicking on any of their past orders, they should be able to see the products which were a part of that order, along with their quantity and price. These orders include snapshots of the product at the time of purchase so if the product was updated after that by the administrator, it won’t be updated here.

Use case narrative:

The goal of this use case is to allow customers to see details about their past orders which they have placed in the system so far in case they want to reference some information about a past order. The actor in this use case is the user. The condition is that the user has previously created an account and has placed at least one order in the system. The use case starts when the user clicks on the “View Orders” button from the “Welcome {{username}}” dropdown menu in the navigation bar.

1. User clicks “View Orders” button
2. System redirects user to a table where they can see the order number, date, total cost and status
3. (Optional) User clicks on any of the rows in the table and sees a new table with the products in that order, their price and quantity as well as total cost of the whole order including the shipping price

The postcondition is that the user has viewed their orders and their details.

2.4 Non-functional requirements

The non-functional requirements describe constraints related to the implementation of the application such as performance, security or reliability requirements. Below is a list of these requirements in my application and a short description of their meaning.

1. The application should be secure

The application should be secure against attacks from internal or external sources. All user passwords that are stored in the database should be encrypted and should never be visible anywhere as text strings. Moreover, it is very important that users who are not authorized for a particular service to be denied access to it. For example, users should not be able to navigate to the administrator's panel and make changes there or unregistered users should not have the ability to place orders in the system. The application will use token-based authentication which means that when users validate their identity, in return they receive a unique access token. The token is attached to every subsequent HTTP request and is used to authenticate them. There are two validations – one in the client which is not very reliable and one on the server. Each request that arrives in the API is inspected and if a valid token is found, the request is allowed. If not, the request is rejected as unauthorized. Moreover, the system should be protected from SQL Injection attacks. That means never constructing SQL statements by concatenating literal text and values coming from the user. SQL commands should be constructed by using placeholders for variable data.

2. The application should have a user-friendly interface

This requirement means that the user interface should be straight-forward, and the users should not spend too much time figuring out how to use the different functionalities. Anyone who knows how to use a basic computer should be able to use this application with ease as well. The error messages and exceptions should be properly reported so that the users can make the required changes. The menus, buttons, inputs and outputs should be self-explanatory and intuitive. This means simplistic and minimalistic design that comes in hand with a great look and feel. Smooth web user experience ensures client satisfaction and increased trust towards the business which in turn means more sales and more profit.

3. The structure of the code should follow good coding standards

In the software development industry, there are certain coding standards that should be followed. Writing high-quality code is not easy but it is necessary in case the application will be extended in the future and further improved. The code should be easy to understand, consistent and properly commented. It should implement the separation of concerns principle and have a well-structured and organized software architecture. The different functionalities that exist within the code should be separated in different files, packages and folders. There are many coding practices which can increase the maintainability of the code so that it can be easily modified at any point in time if needed.

2.5 Activity diagram

An activity diagram illustrates the flow of control in a system. It models sequential and concurrent activities. Below is the top-level activity diagram with the main functionalities in the application. Online customers can choose to browse or search for items and then view a specific item. Then they can add the item to their carts or get back to browsing. The customers can choose to view their carts and update them (modify something) or proceed to the checkout.

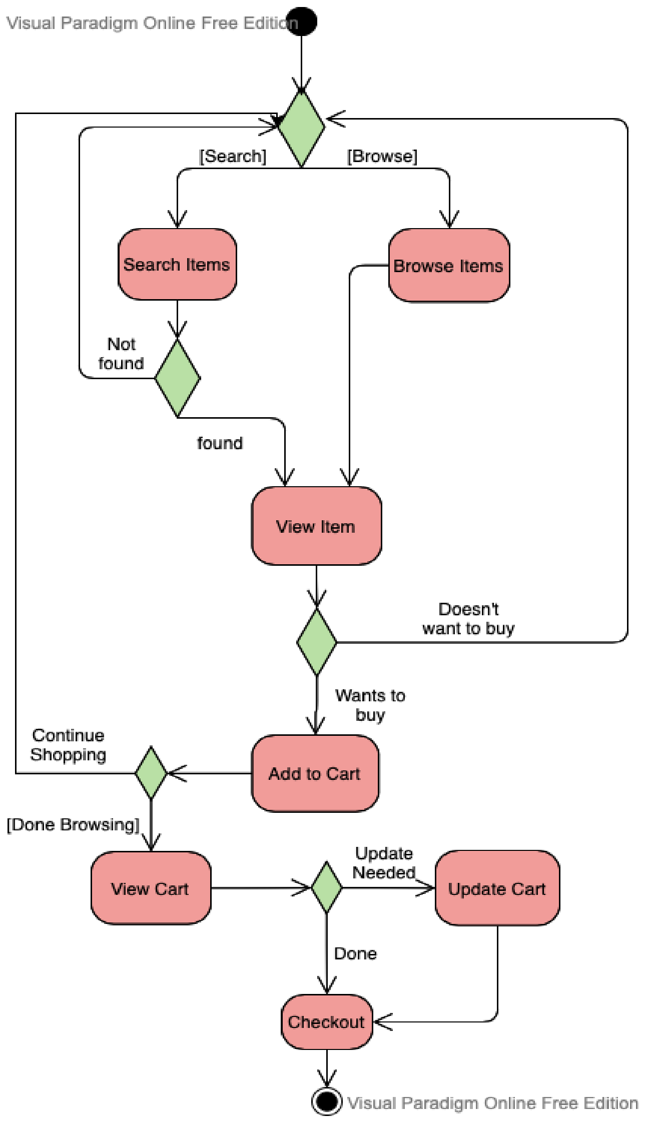


Figure 7 Activity Diagram for the whole application

3. Design of the software solution

3.1 Description of the user interface

As mentioned in the non-functional requirements, the user interface should be intuitive and provide a smooth and pleasant user experience so that there is a clear communication between a user and a computer. When creating the user interface, I have tried to make it clear, consistent, simple and at the same time aesthetic.

- **<u>Structure and Navigation</u>**

There are nine main screens in the application which are: Home, Shop, Product Details, Basket, Checkout, Orders, Admin panel, Login and Registration. Each of them can be accessed through the navigation bar at the top of the screen with the exception of Checkout which can only be accessed from the basket, and Product details which can only be accessed from the Shop page. The navigation bar is a horizontal list of graphical links which is available on all pages of the website. On the left is located the website's logo. In the middle are the menu links for Home and Shop as well as the Admin menu link which is only visible for users with admin privileges. On the right side, there is an icon which serves as a link to the shopping cart. The number above the shopping cart icon shows how many items are currently in there. This number is updated automatically when new items are added or removed. On the right, there is a message which says "Welcome" plus the user's display name which they have provided when registering. When the user clicks on the message a dropdown menu appears with the options to view the basket, view orders and log out of the system. If the user is not logged in, the Login and Sign-up buttons will be located here instead.

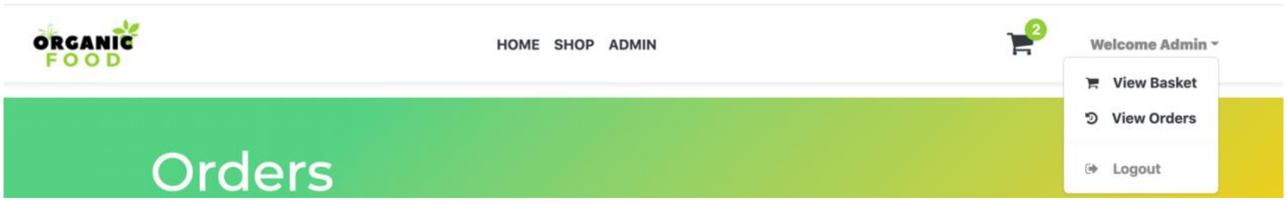


Figure 8 The application's navigation bar

- **<u>Description of the main screen and their component parts</u>**

  - **<u>Home screen</u>**

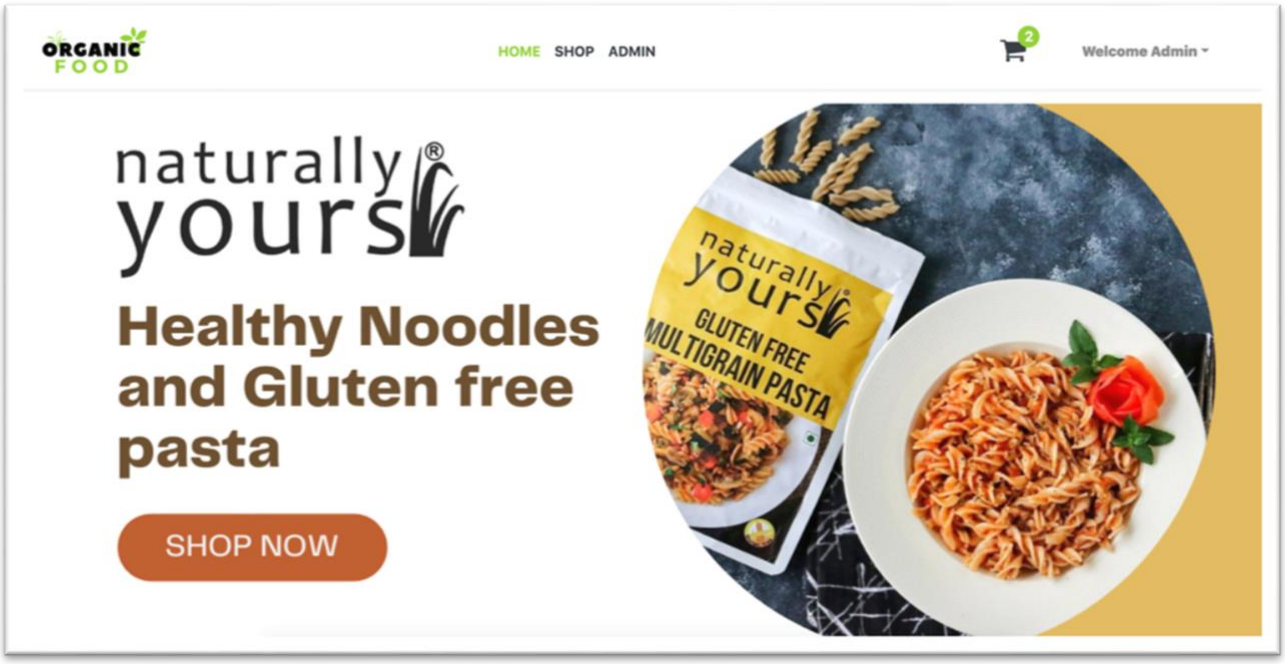


Figure 9 The application's home page

The home page is the entry point of the application. It contains only one element which is a website carousel or a slider. The images change dynamically and display some of the brands offered on the website. The user can also change the images manually.

- **<u>Shop screen</u>**

The shop page enables the users to browse the inventory of products and add them to their shopping carts. The elements on the page can be divided into two categories: the cards with the product details and the helper elements for filtering, sorting and searching which surround the products.

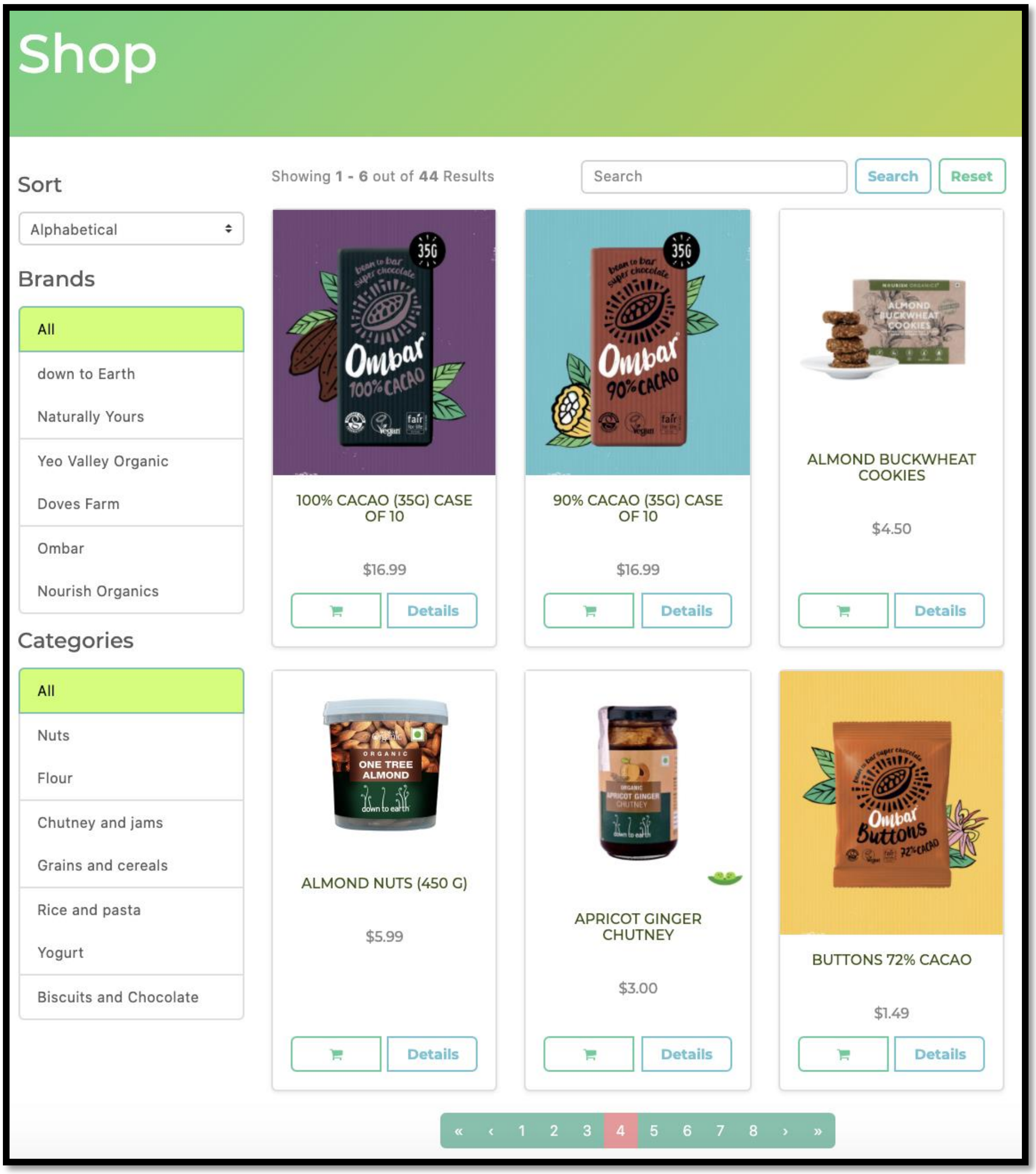


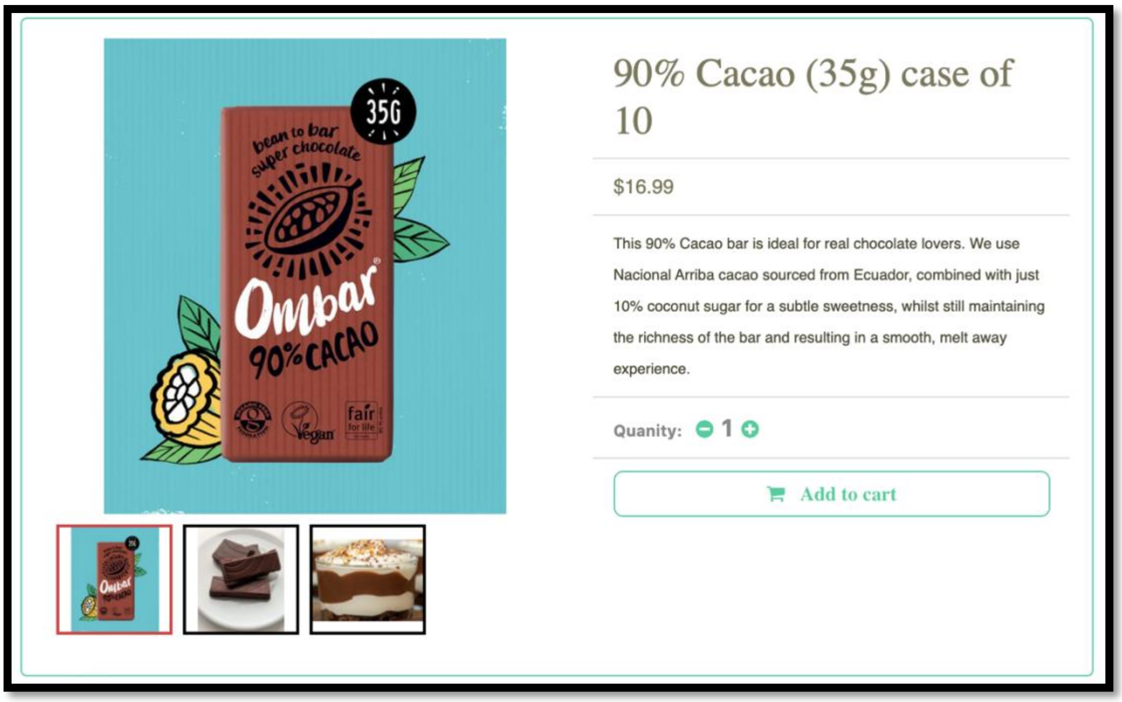
Figure 10 The application's "Shop" page with no filters applied

The main part of the page is occupied by the display of products. The image, price and name are displayed for each of them. Each resides in its own rectangle which are arranged in a grid. There are two available buttons – one for adding the product to the cart and another for seeing details about the product which takes the user to the Product Details page (described in the section below). The items are paginated with only six appearing on the page at a time.

On the top left-hand side is located the Sort dropdown menu which offers three options: Alphabetical, Price: Low to High and Price: High to low. The product list changes dynamically as a new option is chosen. On the right of the "Sort" functionality is the product counter where users can reference how many items they are viewing at the moment and what is the total number of all items. Next, is the Search text box. The users can type in word/s and press the "Search" button. The items that contain the search term or terms in their name will be displayed and the rest of the items will disappear. The "Reset" button resets the search and all items appear again. On the left side of the screen are the two tables with the brands and categories available on the website. The default option chosen for both is "All". When the user clicks on any of the brands or categories in the menus only products from that brand or category are displayed and the name gets highlighted in green so that the users know what they are looking at. On the bottom of the page is the pagination through which the user can navigate the different pages. The current page is highlighted.

- **<u>Product Details Screen</u>**

This page enables the users to see more information about a particular product such as its name, price and description. Here they can also find more images and click on them which makes them larger. Users can also see a number which represents the quantity which can be modified by using the + and – buttons. The "Add to Cart" button adds the item with the chosen quantity to the user's shopping cart.


90% Cacao (35g) case of 10

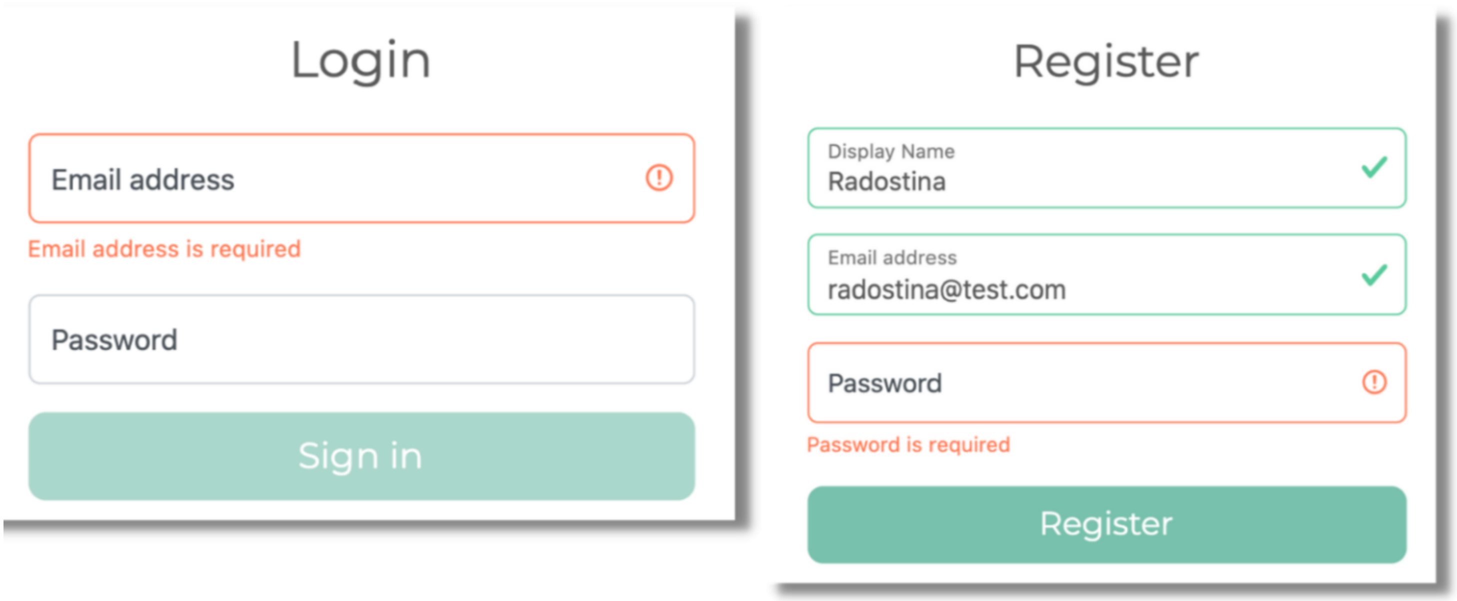
$16.99

This 90% Cacao bar is ideal for real chocolate lovers. We use Nacional Arriba cacao sourced from Ecuador, combined with just 10% coconut sugar for a subtle sweetness, whilst still maintaining the richness of the bar and resulting in a smooth, melt away experience.

Figure 11 Product Details Page for a chocolate bar

○ **Login and Register screens**

The Login and Register forms contain the text fields in which the users have to input the needed information for logging in and registering. If there are errors in any of the fields, they are reported under the text field where the error occurred. The error text color is red so that it can attract the user's attention. The text field is also highlighted in red. If the field has been completed correctly, a green icon appears to indicate that and the text field is highlighted in green. The buttons are disabled until all problems have been resolved.



Email address is required

Display Name: Radostina

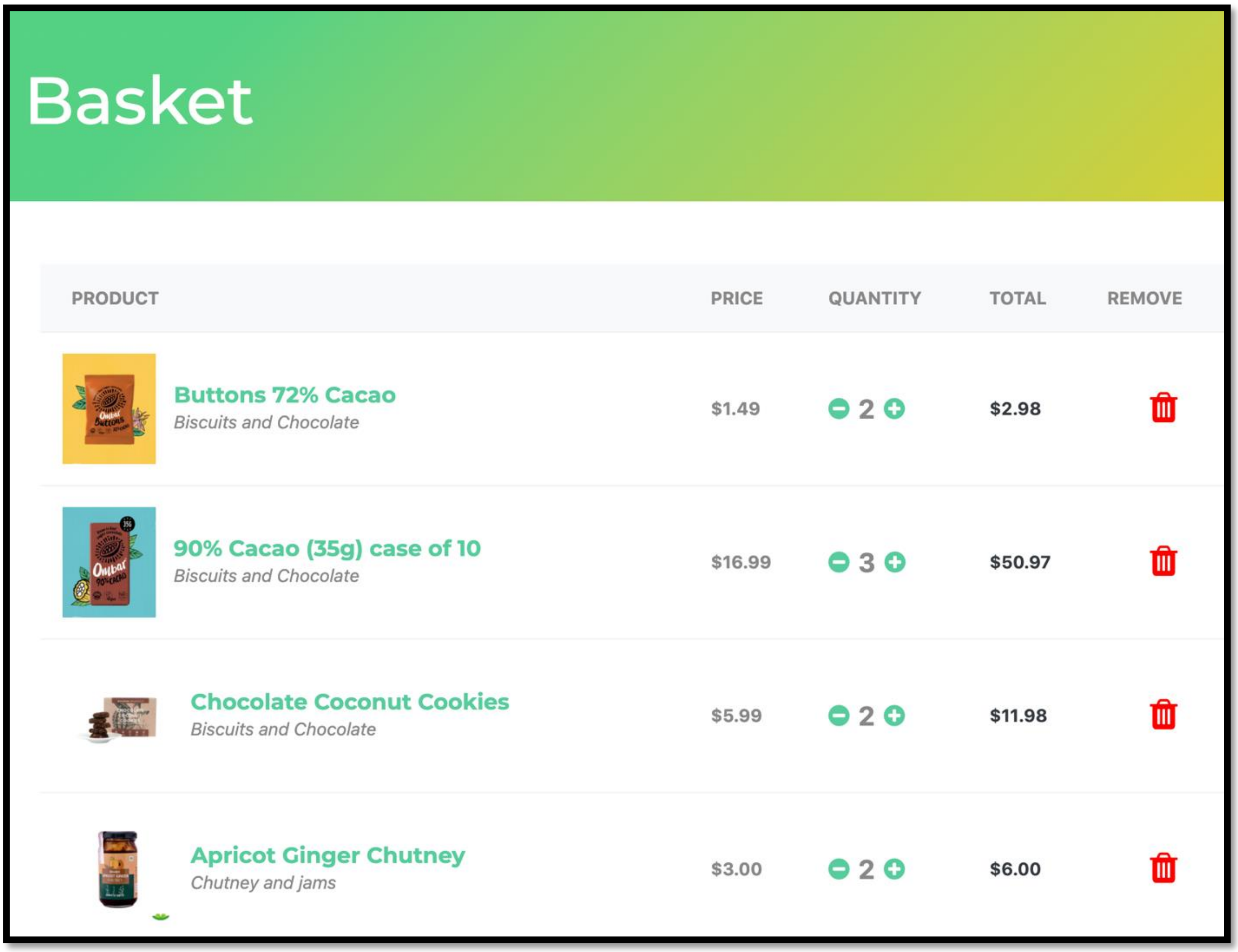
Email address: radostina@test.com

Password is required

Figure 12 Login and Registration Views

○ **Shopping basket screen**

This page enables the users to see the contents of their shopping baskets. It can be reached by clicking on the shopping cart icon in the navigation bar. The products that the users have added are arranged in a table. The table has columns for the product name, price, quantity, total and removing. In the product column, the users can see the image, name and category of each product in their basket. If they click on the name, they will be taken to the Product's details page. The second column is for the price of one individual product. The third column indicates how many of a particular product the customer wants to purchase. The quantity of the products can be increased or decreased by using the plus and minus buttons next to the number. The total price column for each product changes automatically as the users change the quantity. The trash can icon allows the users to remove a product from their shopping basket. At the bottom of this page, the users can find an order summary with the total sum of all of the products currently in their basket. This total changes automatically as the users remove or change the quantity of the products. At the bottom of the Basket page is the "Proceed to checkout" button which redirects them to the "Checkout" page where they can finalize their order.

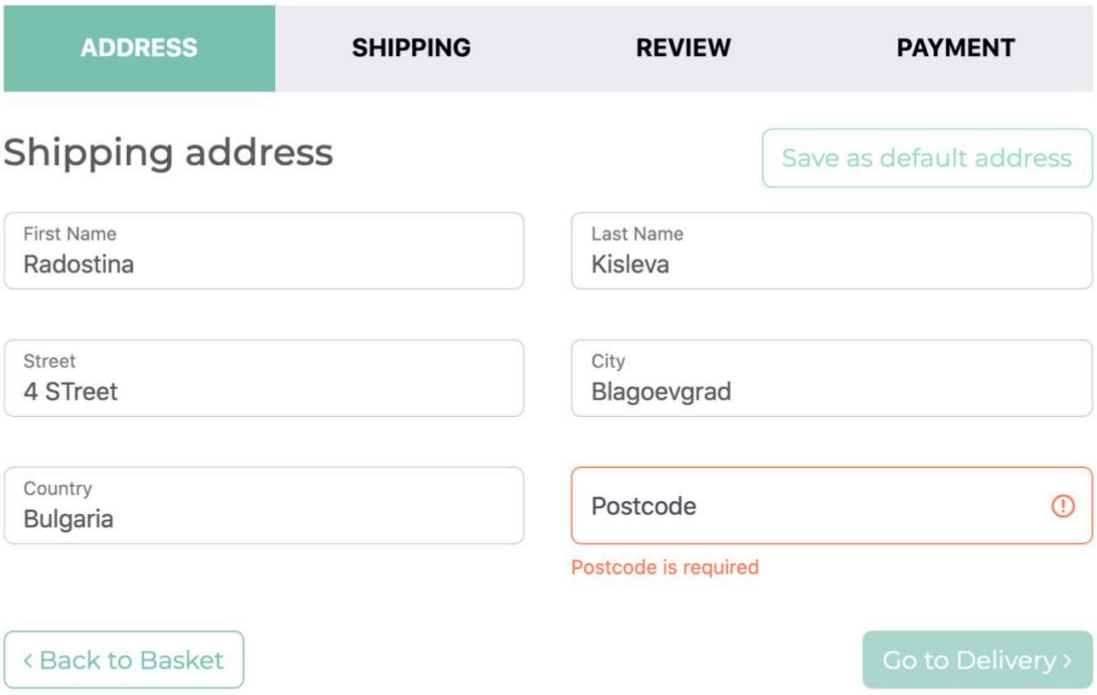

Basket

PRODUCT	PRICE	QUANTITY	TOTAL	REMOVE
Buttons 72% Cacao *Biscuits and Chocolate*	$1.49	2	$2.98	
90% Cacao (35g) case of 10 *Biscuits and Chocolate*	$16.99	3	$50.97	
Chocolate Coconut Cookies *Biscuits and Chocolate*	$5.99	2	$11.98	
Apricot Ginger Chutney *Chutney and jams*	$3.00	2	$6.00	

Figure 13 Shopping basket View

- **Checkout screen**

This page enables the user to complete the checkout process with the information needed to create an order. There are four steps which the user needs to complete in order to place an order. For this purpose, a stepper component is created on this page which has four tabs for the four different steps – Address, Shipping, Review and Payment. The step which the user is currently completing is highlighted in green in the stepper's header. The user cannot proceed to the next step until they have completed the current one. For this reason, the button is disabled and does not allow them to move ahead until they have resolved the issues on this page. The Address and Payment steps are forms which contain text boxes for inputting the required information. If the user inputs the information correctly, the text fields' borders will be outlined in green. If there are any errors, they are reported to the user under using red text the text box which will also be outlined in red. The shipping step displays the four shipping options available. The user is required to choose one of them by clicking on the box next to its name. The review step is simply a list of all products which will be a part of the order. It looks almost identically to the basket view above.



Shipping address

Save as default address

First Name: Radostina	Last Name: Kisleva
Street: 4 STreet	City: Blagoevgrad
Country: Bulgaria	Postcode
	Postcode is required

‹ Back to Basket

Go to Delivery ›

Figure 14 Address form in the Checkout View which contains an error

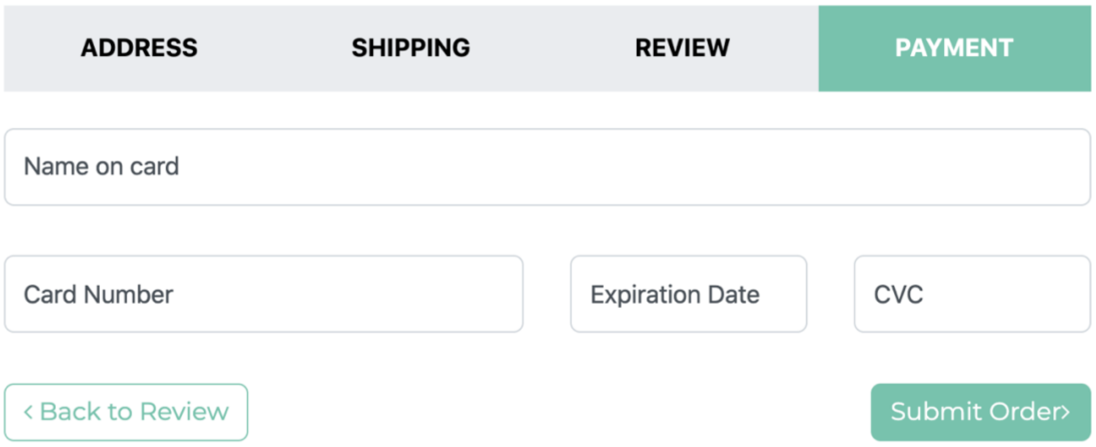
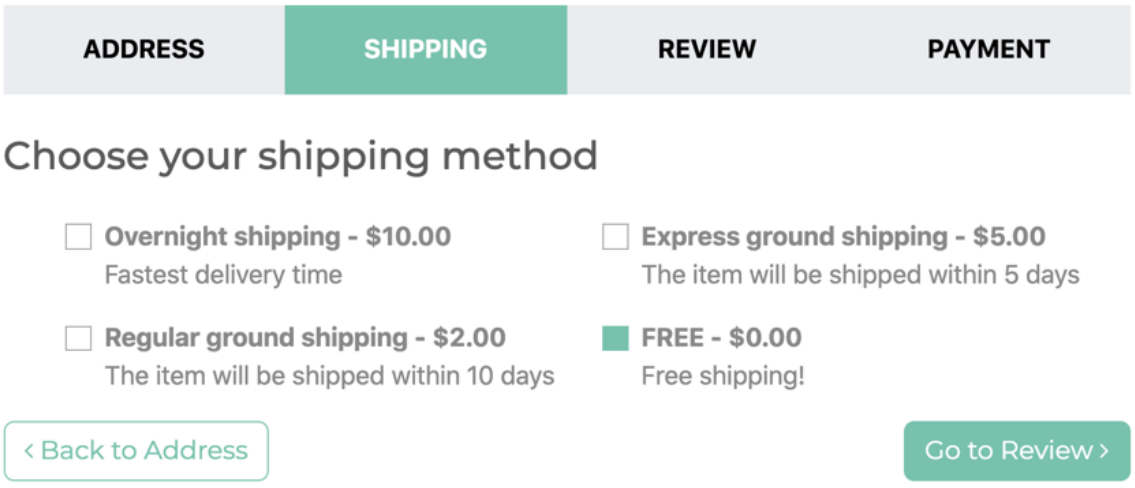


Figure 15 Empty Payment form in Checkout View



Choose your shipping method

☐ **Overnight shipping - $10.00** Fastest delivery time	☐ **Express ground shipping - $5.00** The item will be shipped within 5 days
☐ **Regular ground shipping - $2.00** The item will be shipped within 10 days	🟩 **FREE - $0.00** Free shipping!

‹ Back to Address

Go to Review ›

Figure 16 Shipping Options Display in Checkout View with a shipping option chosen

The checkout page also includes the Order Summary table which is identical to the one in the Shopping Cart View with the exception that the total cost is now updated to include the shipping price which has been added to the subtotal of all products.

Once the user clicks the “Submit Order” button, the following page appears which informs them that their order is confirmed. The “View your order” button takes them to a page which contains a list of all items in the order with their price and quantity, as well as the total price. Also, when an order is submitted, a toast notification appears at the bottom of the page which notifies the user that the order was created successfully.

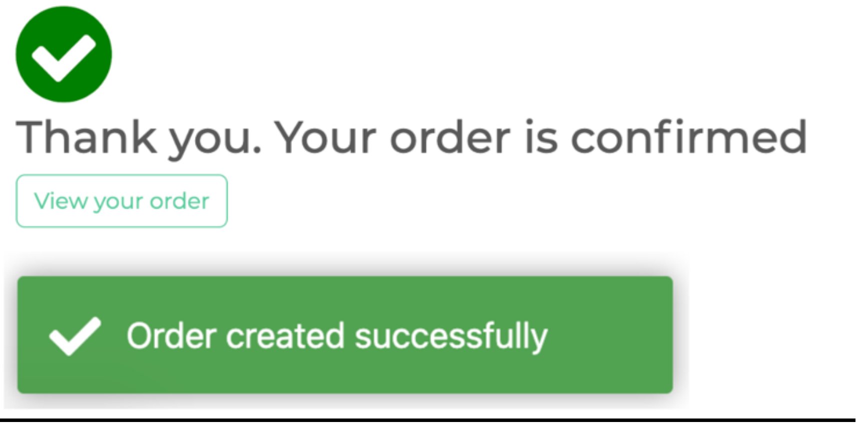
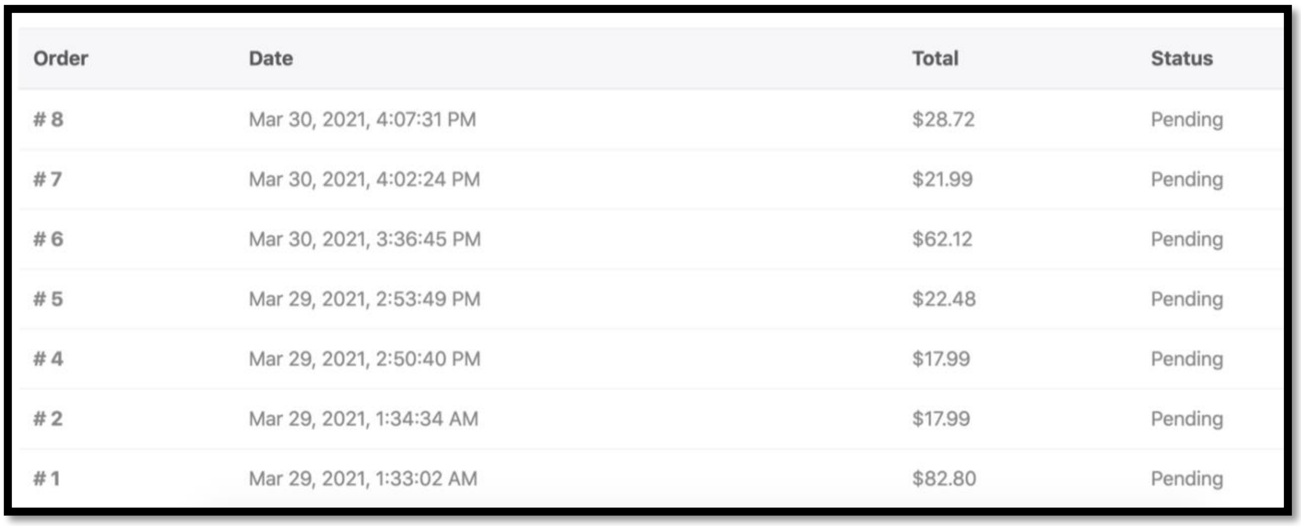

Thank you. Your order is confirmed
View your order
Order created successfully

Figure 17 Order Confirmation View and Toast notification (enlarged)

- **<u>Orders screens</u>**

The orders page can be accessed through the “View orders” menu in the navigation bar. It consists of a table which contains all of the orders the user has made in the system with their date, total cost and status.



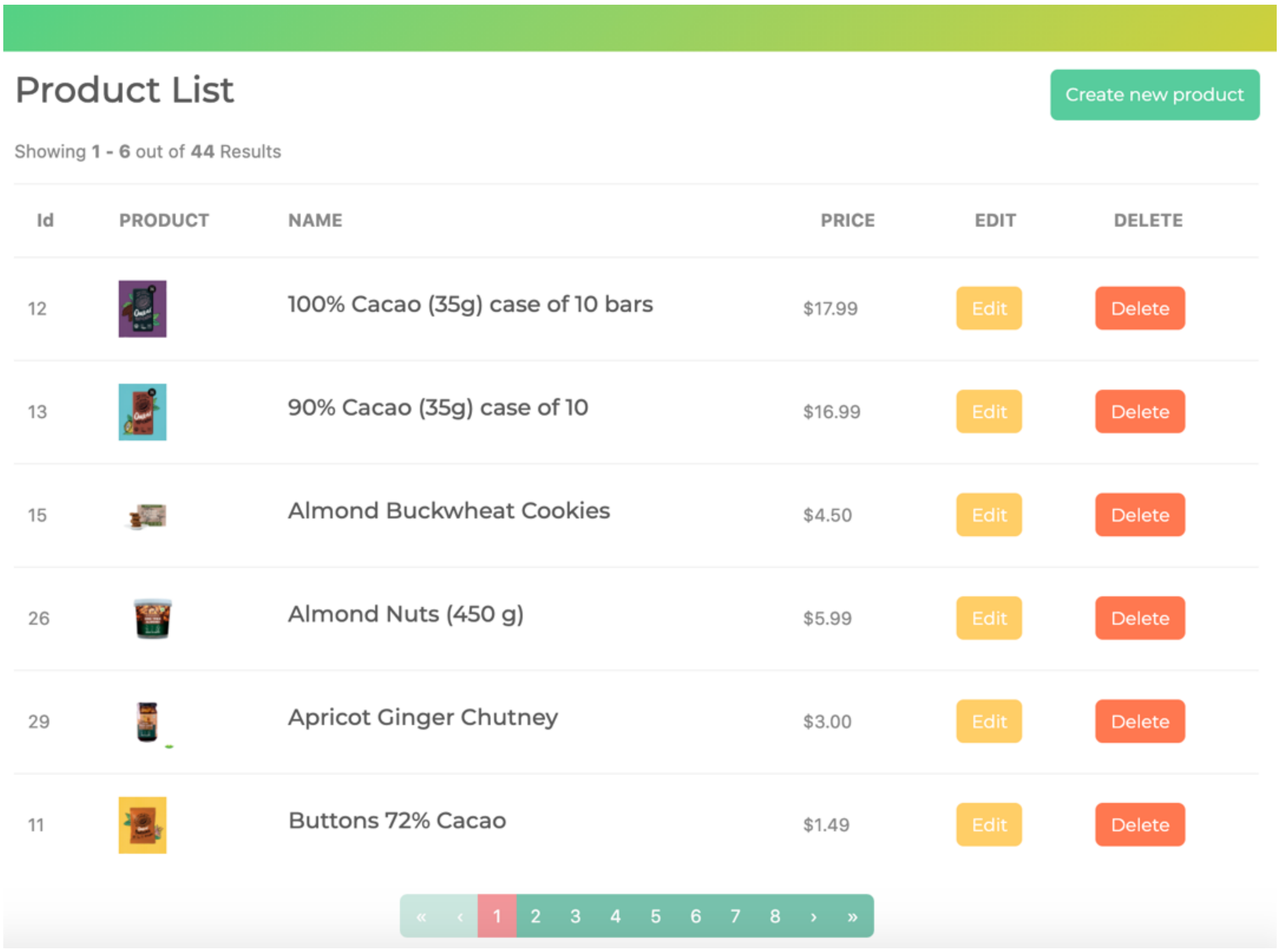
Order	Date	Total	Status
# 8	Mar 30, 2021, 4:07:31 PM	$28.72	Pending
# 7	Mar 30, 2021, 4:02:24 PM	$21.99	Pending
# 6	Mar 30, 2021, 3:36:45 PM	$62.12	Pending
# 5	Mar 29, 2021, 2:53:49 PM	$22.48	Pending
# 4	Mar 29, 2021, 2:50:40 PM	$17.99	Pending
# 2	Mar 29, 2021, 1:34:34 AM	$17.99	Pending
# 1	Mar 29, 2021, 1:33:02 AM	$82.80	Pending

Figure 18 All orders for a user view

When a user clicks on any of those orders, they are taken to the order’s details page which contains a list of all items in the order, their price, quantity, subtotal and total cost.

- **Admin screens**

The purpose of these screens is to allow the administrator to manage the inventory of products. The first one is the Product list screen with a table which contains all of the products currently offered by the store. The administrator can see their ID, image, name and price along with two buttons – the yellow one for editing and the red one for deleting the item. The items are paginated with only six appearing on the page and a pagination navigator under the table. At the top of the table is the green "Create new product" button which takes the admin to the Product creation form. The "Edit" button redirects the admin to the same form but this time the text boxes prepopulated with information about the product which is being edited. The Delete button removes an item from the store.


Product List

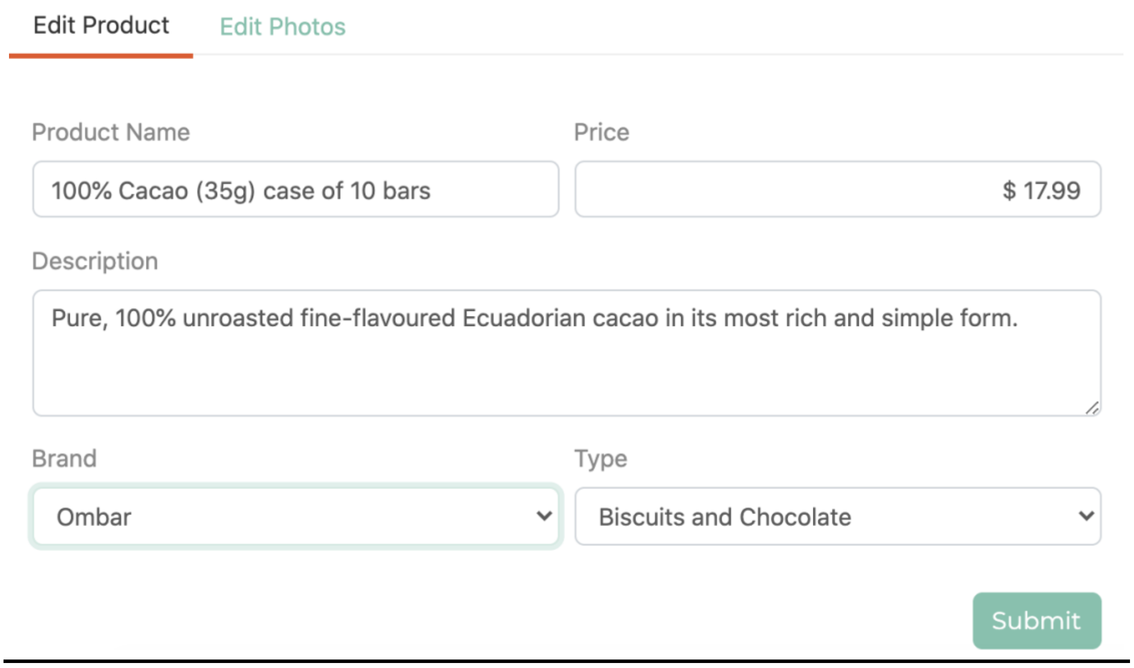
Create new product

Showing **1 - 6** out of **44** Results

Id	PRODUCT	NAME	PRICE	EDIT	DELETE
12		100% Cacao (35g) case of 10 bars	$17.99	Edit	Delete
13		90% Cacao (35g) case of 10	$16.99	Edit	Delete
15		Almond Buckwheat Cookies	$4.50	Edit	Delete
26		Almond Nuts (450 g)	$5.99	Edit	Delete
29		Apricot Ginger Chutney	$3.00	Edit	Delete
11		Buttons 72% Cacao	$1.49	Edit	Delete

Figure 19 Admin Panel Overview

The Edit and Create product forms have two sections which are represented as tabs at the top of the form. The one on which the user is located is underlined with orange and the text color is black. The first tab is for editing the product. There are text fields which can be filled by the admin and two dropdown menus for brand and category. The price text field only accepts numbers. If there are errors, they are reported under the text box with a red text color. If the data is valid, the text box is outlined with green. The green "Submit" button at the bottom of the form is for creating a new product or saving the changes.



Edit Product | Edit Photos

Product Name	Price
100% Cacao (35g) case of 10 bars	$ 17.99

Description

Pure, 100% unroasted fine-flavoured Ecuadorian cacao in its most rich and simple form.

Brand	Type
Ombar	Biscuits and Chocolate

Figure 20 Edit Product Form for Administrators

The “Edit photos” tab displays all images for an item. Below each image there is a button for setting it a new main photo and another one for deleting the image. When the admin chooses to add a new photo, the following screen appears. The Step 1 – Add photo section, contains a dropzone where a new image can be dropped, or the user’s file explorer opens. In Step 2, the image can be resized and cropped. Step 3 displays the preview of how the image will look. The “Upload image” button saves the image.

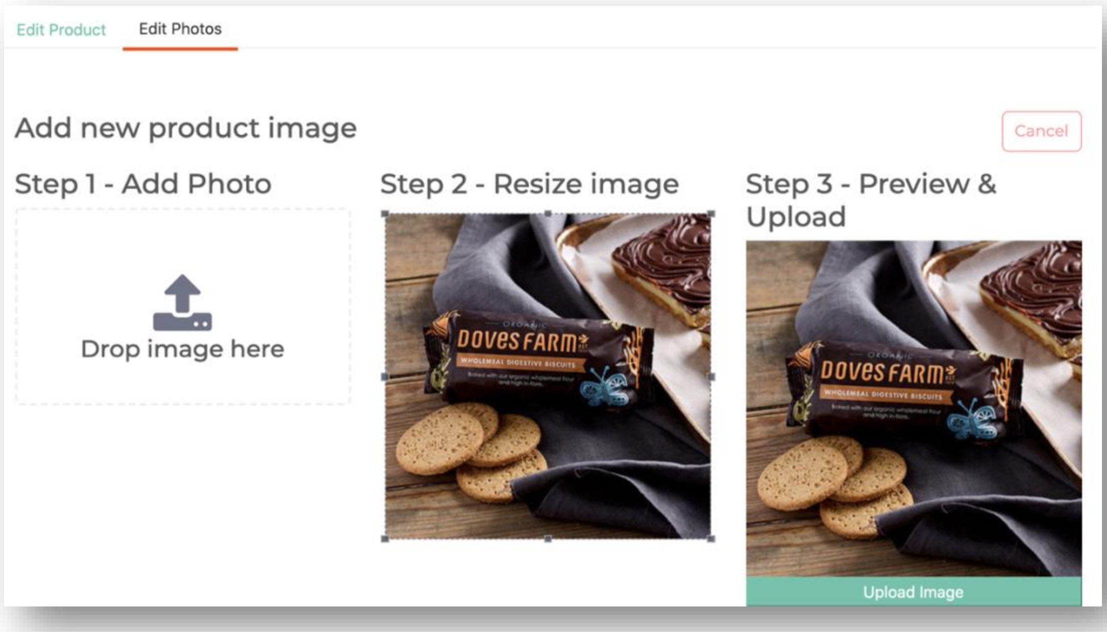


Figure 21 Edit Photos part of Edit Product View

3.2 Software architecture

3.2.1 Overview

Software architecture is the organization of the software system including its components, the relationships between them and the environment. It is concerned with how the code is structured and organized. The different functionalities that exist within the code must be separated and components that share common or similar logic should be grouped together. This is one of the main prerequisites for writing high-quality code.

The most common way to separate the different functionalities withing a system is to group them into different categories – User Interface, Application Logic and Data Access code functions. In order to achieve this, the best practice is to use a defined software architecture in order to separate the related sections of code.

Starting from the top level, the two fundamental components of my web application are the backend and frontend. This separation of the code is the generally accepted practice when building web applications. Frontend, also called client-side, communicates with the backend, also called server-side, via the Internet. The frontend is what is visible to the user. The frontend sends queries to the backend via the middleware [1]. The diagram below is a graphical representation of how the frontend and the backend work together:

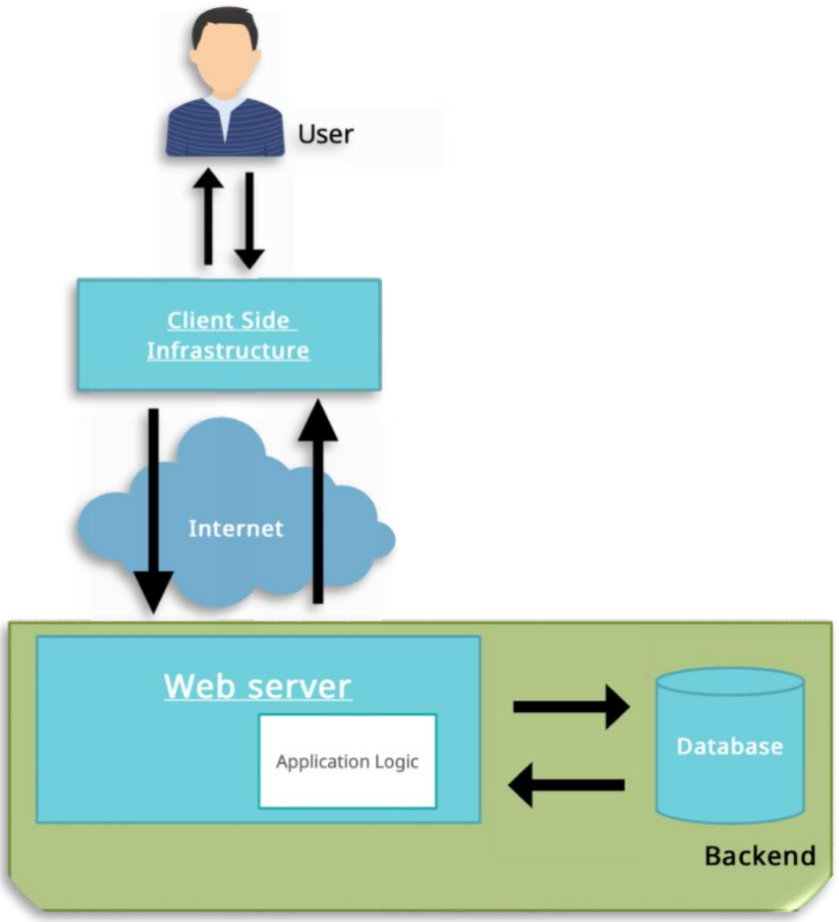


Figure 22 Front and Backend Communication Diagram

The backend protects the data and responds to the queries asked by the frontend. The backend stores persistent data and responds to HTTP requests that run on a server. It cannot be seen by the user and only responds to HTTP requests for a particular URL [2]. The backend interacts with the permanent storage which is the database. It also renders pages to the client and processes user input.

The frontend infrastructure includes everything the user interreacts with. It assimilates the various components that fit together to offer the user interface. It is essentially the code that is parsed by the browser and responds to user input. It can be seen and edited by the user and cannot store anything that lasts beyond a page refresh. It also cannot read files off of a server directly and must communicate via HTTP requests [2]. The script directly interacts with the page’s HTML elements like text boxes, buttons, lists and tables.

The way the two work together is the following: the server delivers the different pages, and handles saving and loading, based on simple messages from the browser. Examples of server-side processing are user validation, saving and retrieving data and navigating to other pages. The actual graphical user interface, all of the reactions to the user’s input and everything related to the visual appearance are completely controlled by the client-side. The server sends raw data – in JSON format to the browser which puts it together nicely and seamlessly.

This separation of the components in this way has numerous benefits. One of them is easier scalability. When the code is divided into two parts, it can be optimized faster. Another benefit is the servers working less and instead of dealing with everything including putting the data in an HTML format, they are only concerned with sending the raw data. As a result, the user experience is better and faster since the whole page is not updating with every request. A third benefit is modularity and working on one module while keeping the others untouched. Keeping the frontend and backend separated decreases the chances of breaking the entire website and makes debugging a lot easier since it is known easily whether an issue is backend or frontend related [3].

3.2.2 Backend Software architecture

The frontend and backend projects have their own architectures which consist of different components and modules. They require different code organization and structure. In the next few pages, I will describe in detail the selected software architectures. Starting with the backend part of the application, I have separated the code into three projects which have different dependencies on each other and are responsible for different things.

I had doubts which software architecture pattern to use. One option was the layered architecture which is used most frequently. However, it creates very tight coupling because each layer is coupled to the levels beneath it and each layer is coupled on some common infrastructure concerns. This makes the application difficult to scale and hinders the growth of the project.

A better option, in my opinion, is an architecture known as Onion Architecture or Clean Architecture, which is the approach that I have followed. This architecture puts the business logic in the center of the application. Instead of having the business logic depend on the data access layer, like in the layered architecture pattern, the dependency is inverted: data access and implementation details depend on the Core. The Core project contains the business logic and is at the center of the architecture design. All other project dependencies point towards it. This is known as the Dependency Inversion Principle. It is achieved by defining abstractions, or interfaces in the Core which are then implemented in the Data Access project. The figure below demonstrates this concept. The Core has no dependencies on the other layers, the API project depends on the Data Access project and the Data Access Project depends on the Core [4]:

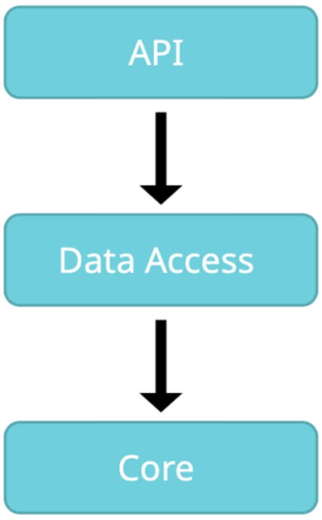


Figure 23 Backend Architecture Diagram illustrating project dependencies

This database is not in the center, as in the Layered Architecture. It is external. The database is used as a storage service through external infrastructure code. Each project has clear responsibilities which emphasizes the separation of concerns principle [5].

Below, I will describe the responsibilities and structure of each of the three backend projects in my application which have been constructed in accordance with the specifications and recommendations given by the Onion Architecture pattern, described above. I describe in detail what packages, folders, files and classes are included in each of the projects.

- **Core Project**

Starting with the Core Project, as described by the Onion Architecture, it is in the center and all other projects point towards it. It contains the business model which includes the business entities or the classes that contain the properties for the business objects. The entities encapsulate the most general rules, and they are the least likely to change. The entities in my application can be divided into four main groups: entities related to Identity, Orders, Basket and Products. The Core project also includes interfaces, such as the generic repository interface, specific interfaces which contain methods needed for performing operations on the entities and interfaces for some of the services. The interfaces only describe the methods that the classes that implement them should support. In other words, they only include abstractions for operations that will be performed in the Data Access project because they involve working with the database. Since the Specification design pattern, which I will describe in detail below, is used, there must be a package with all of the specifications. This package resides in the Core project because the specifications are very unlikely to ever change. They contain the queries to be passed to a particular method for executing some very specific data access requests. Some examples of specifications in the application are getting products from the database with their brands and categories and getting orders with their items, shipping option and date.

- **Data Access Project**

This project communicates with the database and sends queries to get data. Here are the data access implementation classes that implement the repository interfaces from the Core project and are therefore coupled to it. The methods for getting, updating and deleting entities are implemented here. The entity configurations are also stored here. They allow changing and configuring some properties of the entities that have been created by Entity Framework when using the code-first approach. While this can be done in the context class, it is better to do it in separate files. Additionally, here are located the JSON files which Entity Framework will use to seed the database with the products, brands, types and shipping options once the application starts as well as the logic to perform this seeding. Infrastructure-specific services are also located here. These services are implementations of the interfaces in the Core project.

Since this is an ASP.NET web application, an Entity Framework DbContext and a folder with migrations is required and since they are data access implementations, they are located in this project. There are two contexts and two migration folders in the application since the data related to Identity and the user information is kept in a different database and is separated from the data for the products, orders, shipping options, etc.

- **API Project**

The most outer layer of the architecture is the API project. This project receives HTTP requests and responds to them. This is the entry point to the application, and the part that communicates with the client. The controllers which respond to requests by the client are located here. Each browser request is mapped to a particular controller which generates the response to that request. In addition, the folder with the Data Transfer Objects (DTOs) is located here. They allow reshaping the data when returning it to the client such as omitting some properties and flattening objects that contain other nested objects so that the client doesn't see them. It also includes an extensions package which contains static classes that extend other classes. There are also some miscellaneous files such as the Mapping profiles for AutoMapper and some URL resolvers to get the full address of where properties or objects are served from. These miscellaneous files all reside in a folder called Helpers. Another package is the "Errors" one which contains logic for configuring the error responses which will be sent to the client so that it can handle them more easily and also consistently. This project also includes the `public static void Main` method in Program.cs, as well as the configuration systems (appsettings.json file) and environment variables. Here is also the Startup class, which is important for configuring the application, wiring up implementation types to interfaces and allowing dependency injection to work properly [4]. The wwwroot folder is here in which static files can be stored and accessed with a relative path to that root. The folder contains the actual image files for the product's images since only the image URLs are stored in the database.

A visual representation of the backend projects with their components can be seen on the following diagram where the three projects and the main packages that they consist of are displayed. The arrows illustrate the dependencies between the projects:

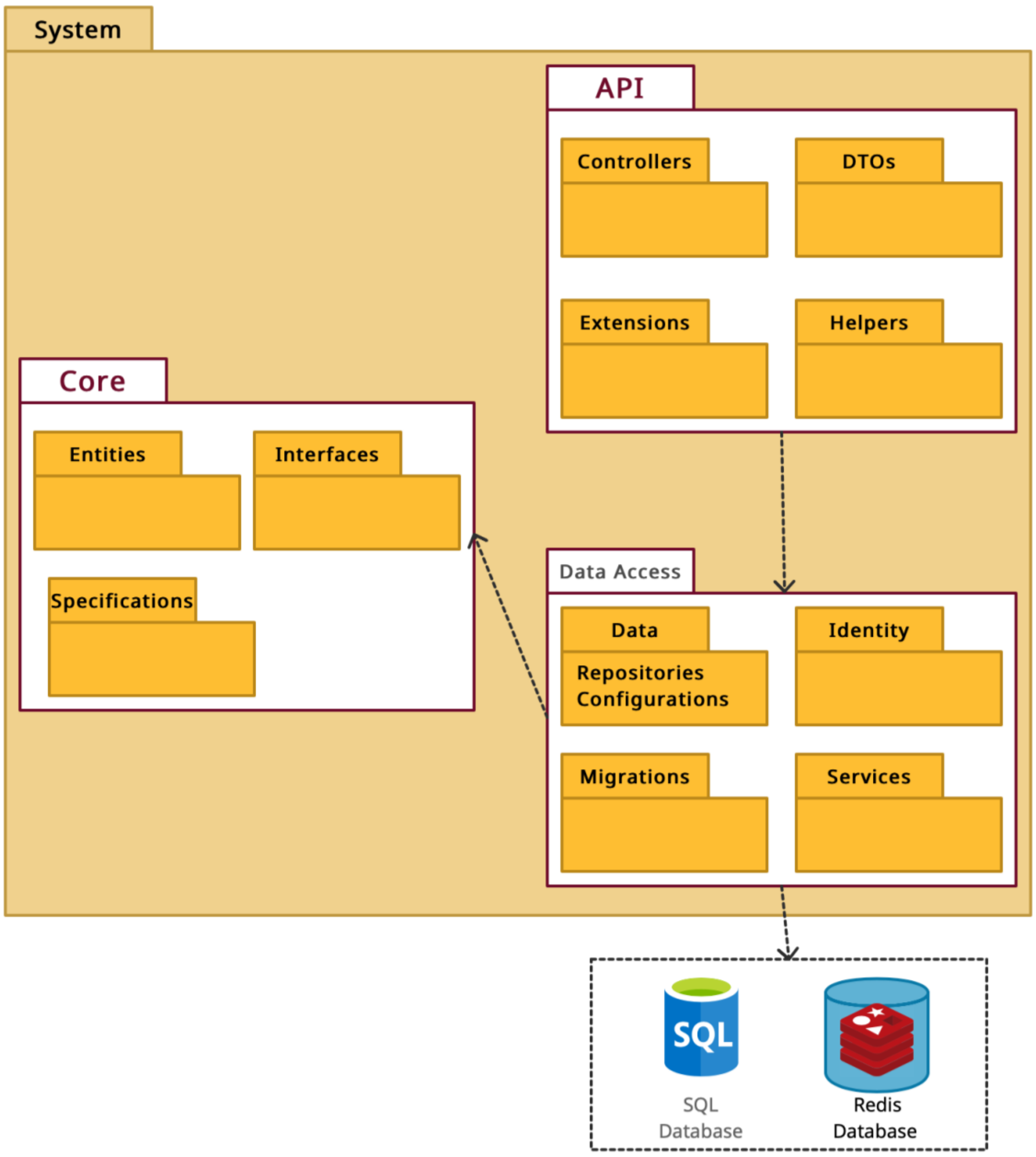


Figure 24 Backend Projects and their Components

A software design pattern is a general reusable solution to a commonly occurring problem withing a given context in software design. It is a template or description of how to solve problems that can be used in many situations. Design patterns are considered best practices that the programmers can use to solve common problems when designing an application or system. A description of the design patterns used for the development of the current web application is given below.

- <u>Generic Repository Pattern</u>

The Repository Pattern is a very commonly used pattern in .NET. It decouples business code from the data access code. Even though the data access code is already decoupled from the database because of Entity Framework, it is not a good practice to put the database context in the controllers. If all of the controllers inject the database context, the result will be a lot of duplicate code because they will all be using the same query logic. The repository pattern implements the separation of concerns principle and adds an increased level of abstraction. The repositories are the layer between the controllers and the DbContext. They are injected in the controllers which use the repository methods to interact with the database. The DbContext will translate the query into a database query and that will get passed back to the controller which will return the results to the client. In this way the controllers become easier to manage and a lot cleaner [17].

While having a single repository is fine for a small number of entities, as the number increases it will be very messy to put all data access logic into one single repository. It will be even messier to create separate repositories for each entity, especially if they all will do the same kind of work. For this reason, it is better to have a single generic repository that can be used with all kinds of different entitles. Very often the operations performed on these entities will be very similar with the exception of the type that is returned so by using generics several, methods can be replaced with just one which helps avoid writing a lot of duplicate code. When using generics, a placeholder <T> is specified which will then be replaced with an actual type at compile time. Using a Generic repository means that there is automatically a repository for every entity.

- <u>Specification Pattern</u>

While the Generic Repository Pattern increases flexibility and maintainability, it makes it harder to optimize certain operations with queries on the database. With the Generic Repository Pattern, the entity is returned as a whole and then it is shaped it after the data has been returned. However, very often the data access requests are very specific which makes them difficult to service using a generic repository. Every time a new query is required, the repository has to be extended with an extra query method. This will result is a lot of duplicate queries [6]. A better thing to do is just send the needed data from the database. The Specification pattern allows for asking the database for very specific query logic. This pattern is very commonly paired with the Generic Repository Pattern. The Specification pattern allows defining specification objects which contain the query that is to be sent to the repository method. The name of the specification reveals its purpose and makes it clear and reusable [6]. The generic methods take a specification object as a parameter instead of a generic expression. Each time a specific subset of data is needed from the database, a specification class is created which allows fetching exactly the specific data that we are looking for. For example, the specification pattern was used when implementing the filtering, sorting, searching and pagination functionalities.

3.2.2 Frontend Software architecture

The architecture of the frontend project was easier to manage since the Angular framework offers many guidelines and recommendations about the structure of the project. The architecture of Angular applications depends on fundamental concepts. The basic building blocks of the Angular framework are components. Components define views, which are sets of screen elements, and services, which provide specific functionality. The components are organized into modules. Modules place related code into functional sets and an Angular application is composed of a set of modules. A module declares a compilation context for a set of components. To summarize, the Angular application consists of HTML templates which are associated with a component class that manage them. The application logic is in services and components and services are boxed into modules. Every Angular application has at least a root module, also named AppModule, which enables bootstrapping, and many functional feature modules. The feature modules represent the different feature functionalities of the application.

Organizing the code into different modules enhances reusability and allows taking advantage of lazy loading – loading modules on demand to minimize the code that needs to be added at startup [7].

In my application, besides the App Module and the modules for each of the main features and functionalities, I also created two additional modules: Core module and Shared Module. Below I will explain the purpose of each of those modules and their structure.

- **App Module**

This module is needed for bootstrapping the application to launch. Every Angular application has at least one module which is this App Module. It is responsible for declaring the components and making sure the components are loaded so that they can be displayed on the HTML page. The app module evolves as the application grows.

- **Core Module**

This module is used for singletons which are services shared throughout the application. The common services are defined in this module and they are only instantiated once. They are always available in the application and any element can use them. This module is imported from the root module only. Below are the components which are a part of the Core Module in the application:

- Category-header – the header available on most of the pages which tells the user on which page they are currently located. Uses the Breadcrumb service which I will talk about at a later point in this thesis

- Authentication guards – the services for handling authentication which are utilized in the entire app. There is an authentication guard which protects the checkout page when a user is not logged in. The is also an admin guard which is used to check if the user is an admin when they are trying to access the admin panel.

- Interceptors – any request that comes back from the API can be intercepted. With interceptors, the request can be intercepted on the way out of the Angular application, and something can be done with the response as it comes into the application. There is a loading interceptor which displays a spinner on the page while the page is loading. The JWT interceptor is used to send the token and if a token exists, it is automatically set to the headers of any request which will be sent to the API.

- Navigation bar - the navigation bar which is located on all pages

- **Shared Module**

This module is for anything that will be used in more than one feature module. Unlike the services in the core module, the ones here won't be available in any part of the application and they will be imported when needed. The models for what standard objects and classes should look like are also stored here because they are used in more than one part of the application. Here reside all the common components that need to be imported inside other modules of the application so that I don't need to import them again and again. Only the Shared module needs to be added to each module to access the common components. These components are:

- Basket Summary component – this component displays the products in the user's basket and includes their name, price, quantity and total cost. It is used in the basket page and the view order page.

- Order-sum – the component which shows the order subtotal, shipping cost and the total cost. It is reused in the basket component, the checkout component and the view order component.

- Pagination header – shows how many items the user is currently viewing and how many items in total are available. It is used in the "Shop" page and Admin panel.

- Pagination navigator – allows the user to switch between the pages and indicates which page the user is currently on. It is reused in the "Shop" page and Admin panel.

- Stepper component – used for each of the steps of the checkout process. The stepper component is adopted from the Angular Component Dev Kit.

- Text-input – this component is shared across the forms in the application and includes the form controls. It is used in order to track the value and validation status of the form controls and display any error messages depending on whether the form is left empty or is populated with wrong data.

- **Feature Modules**

Each specific feature also has its own module with the code relevant only to that specific module. This helps apply clear boundaries for features and keep code related to specific functionality separated from the rest of the code. Each feature in the application has its own module, service, component, template and routing-component. It is be responsible for its own routing since the application uses lazy loading and feature modules are only loaded when they are actually requested instead of just loading them all at once when the application starts. Using lazy loading can significantly improve the loading time and the performance of the application because as the codebase gets bigger and the size of the application increases, it takes longer for the browser to load and parse the source code. Each module has a template file which is a section of HTML. The HTML template renders a view in the browser just like regular HTML but with more functionality. There is special Angular syntax that can extend the HTML vocabulary of the app [8]. Each HTML page has a CSS stylesheet associated with it. The module typically has a related service class which encompasses values, functions and features that the app needs. It can be injected into other components. The component Typescript class (.ts) defines the behavior and the application logic. A component's job is to present properties and methods for data binding and mediate between the view (template) and the application logic. The different services are injected into these classes through the dependency injection principle. There might be different feature modules depending on what the application needs. They are depicted in green rectangles on the diagram below. The diagram also depicts the dependencies between the different moduled. The Core Module is needed in the App Module and that is where it is imported. The Core Module is imported only once. The Shared module is imported in the feature modules if it is needed.

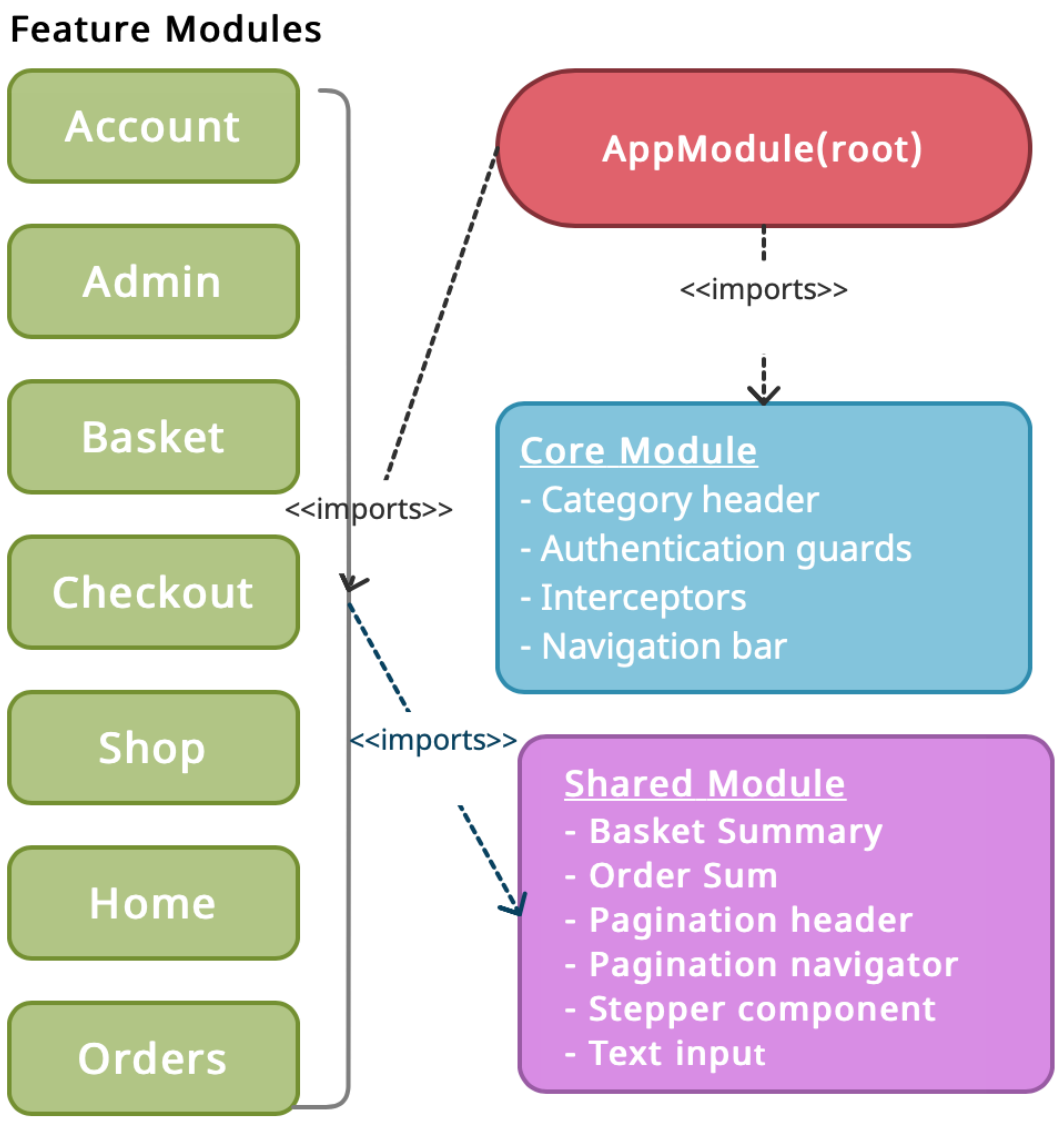


Figure 25 Frontend Modules, their Components and dependencies

For my application, I have identified the following modules for features. Below is a detailed description of each of the modules and what it is responsible for.

- Account feature module – the code related to the logging in and registration features. Includes two components – one for logging in and another for registration. The module has its own routing module.

- Admin feature module – code used for the creation of the admin panel. The components which are a part of this module are the edit product component which consists of the edit product form and edit product photos components. The methods included in the service are for creating, updating, deleting a product, uploading an image, deleting a product photo and setting a main photo. This service is injected into the component classes so that the methods can be used.

- Basket feature module - the code related to the basket functionality. The behaviors available in the service here are for setting the shipping price, getting and setting a basket, adding an item to the basket, incrementing and decrementing an item's quantity, removing an item from the basket, deleting a basket and more.

- Checkout feature module – the components in this module are related to the steps of the checkout process and are called checkout address, checkout payment, checkout review, checkout shipping and checkout success for the success page when an order has been made.

- Shop feature module – the components in this module are the Product Item component, which is the information included in each of the product cards, and the Item Details component which is used in the item's details page.

- Home feature module – the code for the home page of the application

- Orders feature module – includes the Order Details component which is responsible for the logic and display of the user's order.

3.2.3 Data Storage

As I have already mentioned, I am using SQLite as a relational database management system. There are three databases which are used in the application for storing the data. The first one is the primary database which stores the products, the orders, the shipping options, categories, brands and photos. This database is seeded automatically when the application first starts by using JSON files. The second database contains the data connected to Identity which has tables for the users, their addresses and their roles, as well as the other data tables created by Identity. This database uses a different database context and is separated from the other one. When the application is started, two users are seeded in here – one has the normal user permissions and the other has admin privileges. These two databases are controlled by Entity Framework which is the object-database mapper for .NET.

The basket storage is a bit different than the other two databases. When it comes to the baskets, many users often come in the online store, create baskets and then never come back. This would leave way too many unused baskets in the database that will stay there in vain and will possibly never be turned into orders. Moreover, the basket is nothing but client-side state which is not needed once it gets turned into an order. For the above reasons, I do not think it is a good idea to store the basket as permanent storage is an SQL database. Instead, to store the baskets I have used Redis which is an in-memory data structure store. The basket ID and the items in it are stored as key/value pairs. With Redis I am able to set an expiration date on the basket which is going to be erased after a set period of time.

Below are the Entity Relationship Diagrams which are a visual representation of the different entities within the system and how they relate to each other. They describe the logical structure of the database.

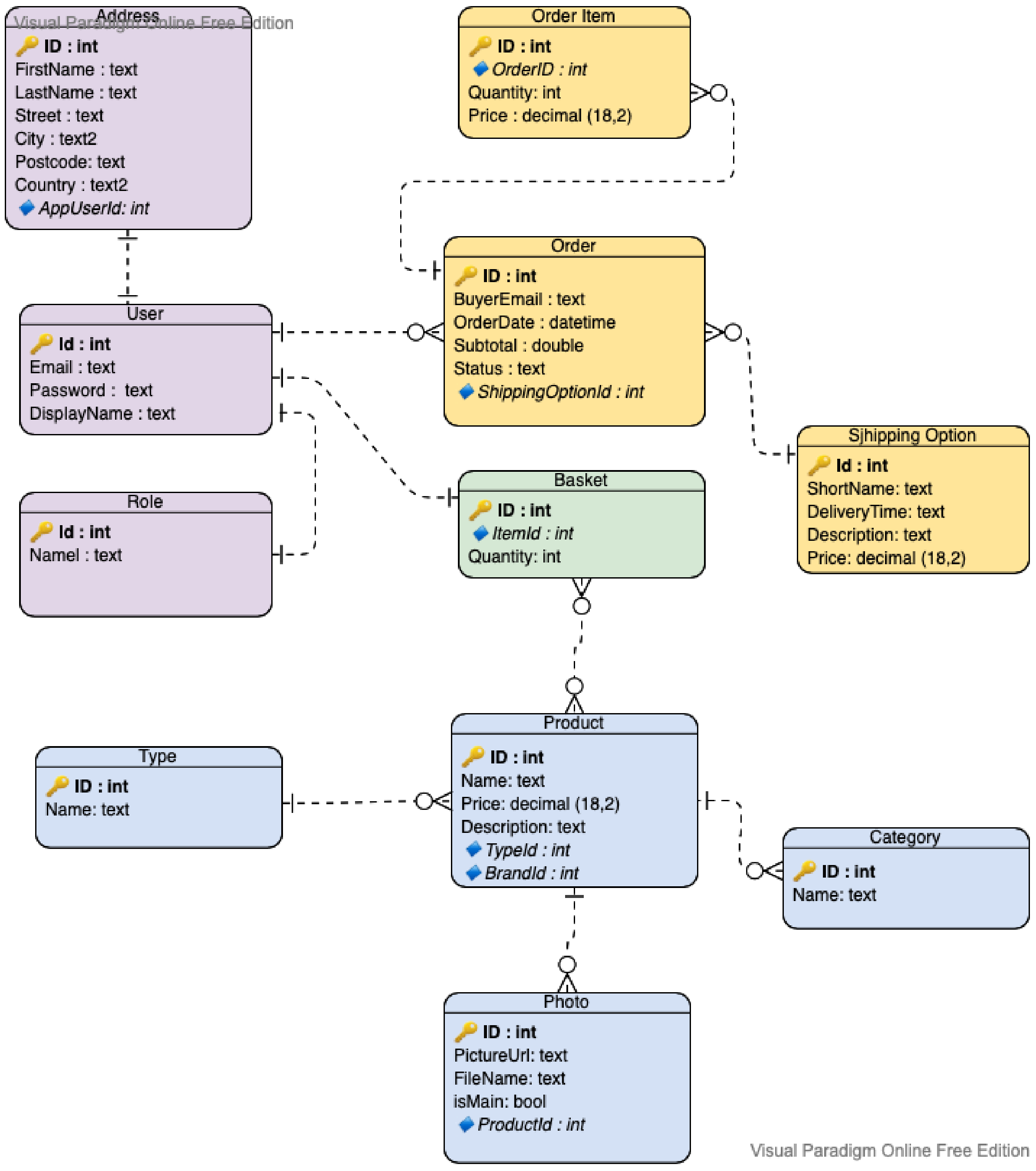


Figure 15 Entity Relationship Diagram

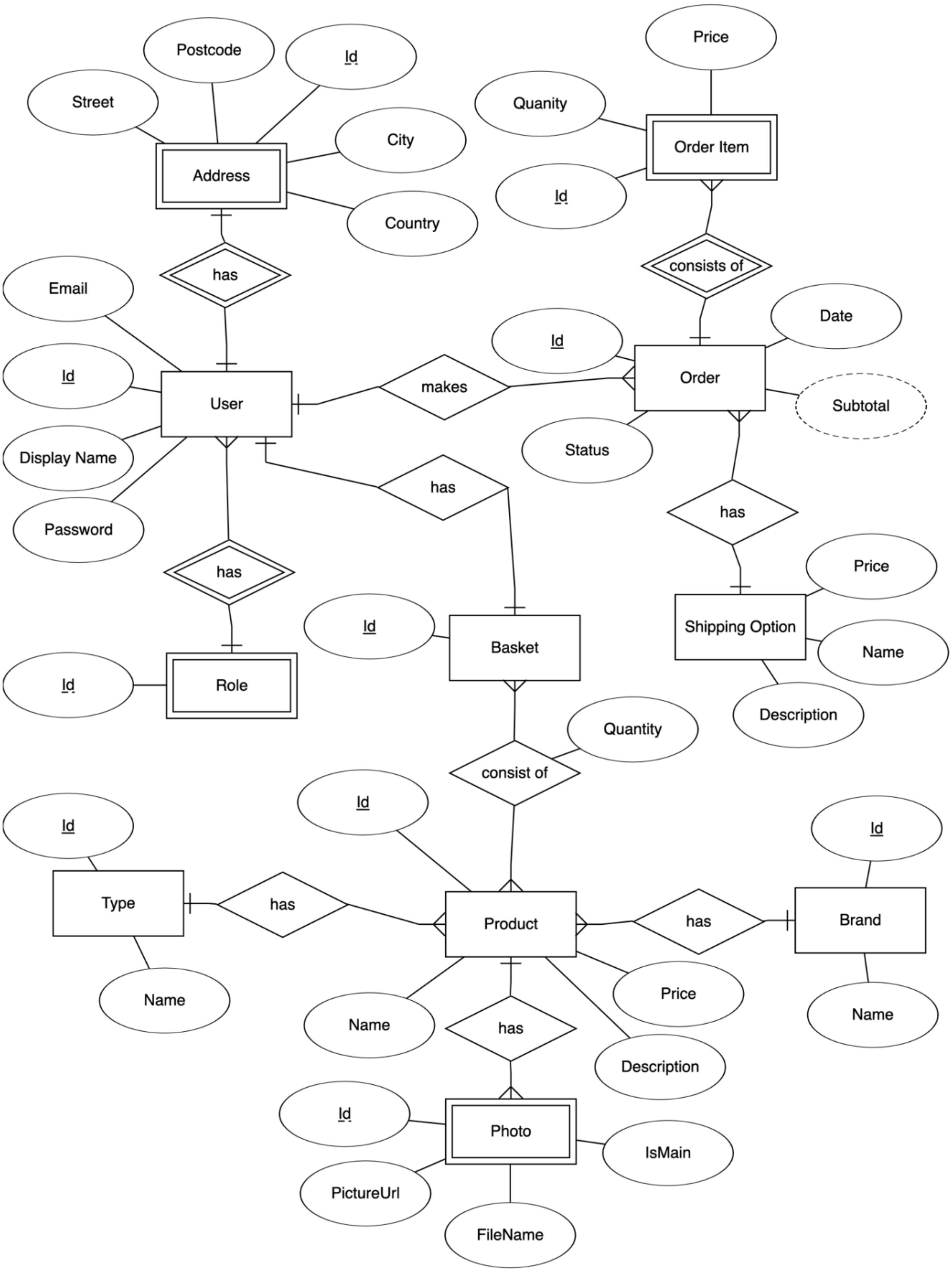


Figure 16 Entity Relationship Diagram

3.3 Security and Reliability

Websites nowadays can get hacked or compromised in many different ways. For that reason, it is important to implement proper security and take the required measures.

To begin with, the website is set to use HTTPS which is a protocol used to provide security over the Internet. It prevents interceptions and interruptions from occurring while the content is transit [9]. I had to configure the Angular project to serve its content over HTTPS. The connection should be encrypted because it asks users to register and sign-up. In the future, they will also be adding credit card information so they need to know that they are interacting with a proper server and nothing else can alter or intercept the connection. Using HTTPS is very important because without it, a hacker might steal the users' personal and financial information.

Another security measure taken is making the users choose strong passwords and hashing them as they are stored into the database. The passwords should be at least 6 characters long and they should contain at least one uppercase and one lowercase character, one non-alphanumeric character and one number. Complex passwords require a lot of time and computational resources to crack. By default, in ASP.NET, Identity is going to hash and salt the user's password. This is done so that passwords are not stored as clear text in the database in case it gets compromised. The password will be hashed using a secure algorithm (PBKDF2). The salt is another random element applied to the hash and it uses 10 000 cycles of sorting that password so it will be very difficult for hackers to get the actual password [10].

The most time-consuming security measure to implement was implementing proper authorization and limiting access to endpoints that certain users are not allowed to visit. For example, only logged-in users are allowed to get to the checkout page and only admins are allowed to get to the admin panel. I decided to use JSON Web tokens for authentication and authorization services. When the users verify their Identity, in return, they will receive a unique access token which will be attached to any subsequent HTTP requests they make. During the lifetime of the token, users will not be asked to re-enter their credentials. JWTs are a method for representing and transmitting claims securely between two parties. The information can be verified and trusted because it is digitally signed. A JWT consist of three parts: header, payload and signature. The payload contains the claims which are statements about an entity. The user's email, name and role will be included in the claims. The claims will be decoded by the client. This information can be read by anyone. Only the token's signature is encrypted which verifies that integrity and confidentiality of the token [11]. Each request coming in the API will be inspected and if a valid token is found, it will be allowed. Certain endpoints of the website will be protected and users who do not send a valid token to the server that passes the checks will not be allowed access to the endpoint.

Last but not least, data validation is very important. Validation attributes should be applied to the data that is received from the client in order to make sure that the API does not accept bad data and also that it returns errors to the client. Validation should be performed on both the browser and server side. The client can catch simple failures like, for example, making sure that required fields are not left empty so that we don't load the server and waste its resources with such small tasks that can be performed by the client. But these can be bypassed so it is more important to validate on the server because malicious code could be inserted into the database which can have very bad consequences. Examples of some of the things against which I am protecting are preventing the client from being able to send empty fields, registering with a bad email address, registering with an email that already exists, basket's items' quantities are not allowed to be set to 0 and more. The validations for the input fields are done at the level at which the user passes the information which in my case are the Data Transfer Objects set for the forms, instead of on the entities. In Angular, validator functions are added directly in the form control model in the component class. Angular calls these functions whenever the value of the control changes. Error messages are important so that the users know how to fix their mistakes. However, not too much information should be given away in the error messages to ensure that nothing on the server is exposed. Messages should be kept as vague as possible.

4. Implementation

4.1 Technologies used

4.1.1 Computing platform and IDE

The project was developed on a computer running Mac OS X version 10.15. It was tested using the browsers Google Chrome and Safari.

The integrated development environment which I have used is Visual Studio Core both for the backend and frontend. The main advantages that drew me towards it were that it is free and that it is cross-platform. The experience on the different operating systems which VS Code provides is almost identical and since I am using Mac OS for the development, using Microsoft's Visual Studio would have been very difficult and uncomfortable. Luckily, Visual Studio Code can be set up to perform like an IDE with the help of some extensions and can be turned into a good editor for cross-platform C# development. It supports features like debugging, IntelliSense, autocompletion syntax highlighting, bracket matching and many more. It is not necessary to use the same IDE for the backend and front project but luckily Visual Studio Code is also a great code editor for Angular. It provides a lot of support for working with TypeScript which is the programming language Angular is using. There are numerous extensions available which can help with Angular development.

An additional software which I have used throughout the development, besides an IDE is Postman. Postman allows easy and quick testing of the API endpoints and methods. It allows creating simple and complex HTTP requests, saving them in collections and reading their responses. All of this results in more efficient and less tedious work.

In order to develop an application using the .NET software development framework, the .NET software development kit (SDK) is needed. The .NET SDK is a set of libraries and tools that allow developers to create .NET applications and libraries. The version of .NET I am using is the latest recommended available which is .NET 5 which will be the main implementation of .NET going forward. Microsoft has dropped the "Core" from the name of the framework. The object-database mapper for .NET is Entity Framework. The version of Entity Framework which I am using is 3.1.10. Throughout development, I have also used the .NET command-line interface (CLI) which is a cross-platform toolchain for developing, building and running .NET applications. It is included in the .NET SDK.

When it comes to Angular, first and foremost, the Angular command line interface (CLI) is needed. The Angular CLI is used to create projects, add files, perform updates, deal with debugging, testing and deployment. It makes it easy to create an application that already works right out of the box. The version of Angular I am using is Angular 11.0.7. The Angular CLI and Angular itself need a JavaScript engine in order to run, build and compile an Angular application. What is used is Node.js which is a JavaScript runtime build on Chrome's VB JavaScript engine. The version of Node I am using is 14.15.4.

4.1.2 Programming Language and Frameworks

I have used .NET and C# for creating the backend of the web application. While .NET is the developer platform with the tools, programming languages and libraries for building many different types of applications, the framework for building web applications specifically is ASP.NET. It is a free, stable, open-source and cross-platform and comes with many features. The applications can be run on Windows, Mac and Linux and can be deployed on cloud and or run-on-premises. ASP.NET is said to perform faster than any popular web framework and offers many features out of the box such as security and data validation. It supports industry standard authentication protocols which protect the applications against cross-site scripting (XSS) and cross-site request forgery (CSRF) which is very important. The applications developed with ASP.NET can also easily be integrated with other libraries and databases [12].

Besides web applications, ASP.NET is also used to build secure REST APIs which can be consumed by a broad range of clients which is what I have used. ASP.NET offers many advantages such as simple serialization which means that endpoints automatically serialize classes to properly formatted JSON out of the box without any configuration required. It also offers policy-based authorization and security of endpoints using JSON Web Tokens which I have made use of. The routes can also be defined inline with code which is very convenient.

The programming language used for the backend is C#. It is general-purpose object-oriented programming language. It was developed by Microsoft and requires the .NET framework to work. It is the framework the language is built on. C# is a particularly strong language for building web applications. Working with C# offers many benefits, one of them being the abstraction away of many of the complex tasks so developers do not have to worry about them. .NET 5 applications have access to the latest nineth version of C# which has brought some new features to the language.

As already mention, the framework I am using for the client side is Angular. Angular is a development platform which includes a complement-based framework for web applications, a collection of libraries that cover a wide variety of features and a suite of developer tools to help develop, build, test and update code. Angular is designed to make updating as easy as possible [15]. The Angular framework is component-based and uses declarative HTML templates. It comes with many tools right out of the box such as routing and a preconfigured environment. Some useful features of Angular which I have used are Agular’s modular architecture, lazy loading, routing, forms, creating reusable form components and much more

Angular is built entirely in TypeScript. TypeScript is an open-source language which builds on JavaScript by adding static type definitions. Types provide a way to describe the shape of an object, providing better documentation and allowing TypeScript to validate that the code is working correctly. The best part is that all valid JavaScript is also valid TypeScript code. The type-checking errors (if any) won’t prevent the resulting JavaScript from running. TypeScript just provides for stricter behavior but still the developer is in control which is very useful when developing enterprise-scale applications when developers need to make sure the code is clean and verify its quality more often [13].

I chose to use Angular because it comes with a range of useful features such a vast ecosystem and also tons of ready-made components that make scaling a project easy. I also like Angular’s component-based architecture in which there is a strict hierarchy of the components. Each component in Angular can be thought of as a small piece of user interface, encapsulated with its own functionality. Components of similar nature are well encapsulated and can be reused among different parts of the application. For example, the Order Summary component was reused three times on three different pages throughout the application. The component-based architecture also ensures there is a small coupling between the components which means that they can easily be replaced with better implementations if needed.

4.1.3 Data storage and management

.NET uses Entity Framework Core which is a modern object-database mapper for .NET. It supports LINQ queries, change tracking, updates, schema migrations to interact with databases. It supports many database providers some of which include SQLite, SQL Server, MySQL and some non-relational stores such as MongoDB, Redis and more.

The database engine which I am using is SQLite. The main reason why I chose to use it is because it is very reliable, lightweight and cross-platform. Visual Studio Core offers extensions to query and explore SQLite databases. The Structure Query Language (SQL) can be used to manage the database. I used the code first approach which means that the database columns, tables and data are defined by code and are created from scratch when the application runs. I configured some of the properties of the entities before the database is created.

On the other hand, Redis, which is used only for storing the baskets, doesn’t use the structured query language, but instead comes with its own set of commands for managing and accessing data. Keys serve as unique identifiers for their associated values. The database is managed through the `redis-cli` or the Redis command line interface through the terminal.

4.1.4 Additional libraries and frameworks

To make styling of the project easier and less time consuming, I have used Bootstrap. Bootstrap is a powerful toolkit - a collection of HTML, CSS, and JavaScript tools for creating and building web pages and web applications. It is a free and open-source project which is very flexible and easy to work with. Its main advantages are that it is responsive by design, maintains wide browser compatibility, offers consistent design by using re-usable components, and is very easy to use and quick to learn. In addition to that, it offers rich extensibility using JavaScript, coming with built-in support for jQuery plugins and a programmatic JavaScript API. Bootstrap can be used with any IDE or editor, and any server-side technology and language [16]. It was more important for me to focus on the development work rather than designing the web page so using Bootstrap was extremely useful for me because I was able to create a good-looking design without too much work. There are also free pre-made whole themes for Bootstrap available on Bootswatch. The theme which I used is called “Minty”. Since the website is related to organic food, I thought the main colors of the color scheme should be green hence I decided to use this theme.

Traditionally, jQuery was used for building Bootstrap. It is not common to use a library like jQuery when using Agular. The primary reason is the fact that jQuery is a DOM manipulation library and Angular uses the DOM to track changes and react to those changes. Because of that, when using jQuery, it is tricky to get Angular change detecting working and reacting to those components. Luckily, there exists an Angular specific package for Bootstrap that gives Bootstrap’s functionality in an Angular way. The tool is called ngx-boostrap. The navigation bar, carousel on the home page, dropdown menus, pagination navigator, image cropper, image drop zone and many more are all components which I have adopted from ngx-bootstrap.

To get the vector icons used throughout the website I have used Font Awesome. It is a font and icon toolkit based on CSS. It was made for use with Bootstrap and has been incorporated into the Bootstrap CDN. The icons are of very high quality and can even be styled so that their color and size, can be changed and even animation can be added to them. Some of the icons which I have adopted from here are the basket icon, icons for the buttons, icons for the navigation dropdown menu and more.

There are many libraries which extend Angular’s base functionality. Below I will list some of those libraries which I have used.

- Reactive Extensions for JavaScript (RxJS) – a library for reactive programming using observables that makes it easier to compose asynchronous code. The library provides utility functions for creating and working with observables. It allows setting up multiple channels of data exchange to ease resource consumption which means handling events independently in parallel and continuing execution without waiting for some events to happen and leaving a web page unresponsive [14].

- Angular Forms – what is used in order to handle user input. Angular offers two approaches when it comes to handling user input: reactive and template based. I have used both in my application. The difference between the two is that reactive forms are more scalable, reusable and testable.

- Angular Component Dev Kit – a set of behavior primitives for building UI components. It is a library of predefined UI components. It contains logic for many features which can be included in Angular applications. The component which I have used from this library is the CDK stepper for the Checkout forms. A stepper divides content into logical steps and requires the user to complete previous steps before proceeding on the next one.

- Router – to handle navigation from one view to the next, the router enables navigation by interpreting a browser URL as an instruction to change the view. The routes in Angular are added in a Route array in the Routing module. They contain two properties – the first one for the URL path for the route and the second one for defining the component that should be used for the corresponding path.

- Currency Mask – in order to make sure the price input fields are formatted as currency. This is used for the price input field when the admin is adding a new product.

- Ngx-gallery (by Kolkov) – a simple native gallery component. Used for the gallery of images for each of the products. The gallery is added to the product's details section. It is straightforward and easy to use. The gallery needs to be provided with some options (width, height, thumbnails, animation, preview, etc) and the images which are to be used.

- Ngx-Spinner – a library with more than 50 different loading spinners to choose from. The spinners appear while the information on the page is loading. They are animated and are intended to inform the user that the application is loading. Their style, type, text can be customized.

- Ngx-toastr – toastr is a JavaScript toast notification library for non-blocking notifications. It is used to inform the user when a request is successfully submitted or not. For example, I have used it to inform the user that an order has been placed successfully and also to inform them that some error has occurred. These notifications are not static, and they include animation.

- Xng-breadcrumb - a breadcrumb is a navigation scheme that revels the user's location on the website. It is horizontally arranged on the top of the page below the navigation bar and above any other page content. By using the library, breadcrumb labels are auto generated by analyzing the Angular Route configuration. The breadcrumbs can be updated dynamically, disabled, customized and styled.

4.2 Installation Requirements

In order to run the project, several things need to be installed on the machine that is running it: the dotnet SDK, the Angular CLI, Node JS and a Redis Server.

First, open the project's folder in the terminal and run the `dotnet restore` command. It will restore the dependencies and tools of a project. Make sure the Microsoft.EntityFremewrokCore.Sqlite package is present. Make sure its version matches the version of the SDK. When you open the appsettings.development.json, you will see that there are two SQLite connection strings so there will be 2 databases that will be created when the app is started. There is also a connection string for Redis. Redis needs to be installed locally on the computer so that it can be run. Otherwise, there will be an error. In order to start the Redis server, navigate to the project's folder

using the terminal and run the `redis-server` command. The Redis server will be started. If it is working correctly, when the `redis-cli` command is run inside the project's top-level folder and then the `ping` command is sent, the server should respond with a PONG.
The existing two migrations folders in the DataAccess project should be removed and two new ones should be crated. The following directories should be removed:

DataAceess/Data/Migrations
DataAccess/Identity/Migrations

Then run the following commands using the dotnet CLI (the commands should be run in the folder of the solution):

```
dotnet ef migrations add Initial -p DataAccess -s API -c StoreContext -o Data/Migrations
```

```
dotnet ef migrations add DataAccess -p Infrastructure -s API -c AppIdentityDbContext -o Identity/Migra
tions
```

Now, there should be 2 new migrations folders that use SQLite. The solution is run with the dotnet run command in the API folder.

When it comes to the Angular project, all that needs to be done is run "nmp install" and then inside the client folder run "ng server" which starts the Angular application.

Navigate to the app which is running on https://localhost:4200.

When the database is created for the first time, two new users are seeded into it. One has an admin role and the other has an ordinary member role. Their login credentials are the following:

Admin. email: admin@test.com password: Pa$$w0rd
Member. Email: user@test.com password: Pa$$w0rd

You can use them to login the application or register a new user. However, this is the only admin that will exist in the application since registration only works for ordinary users.

The tables with products, brands, types and shipping options will also be seeded with data coming from the json files in DataAccess/Data/SeedData folder.

4.3 Code fragments

Below I will include some code fragments of implementation features.

The first code fragment depicts the code which returns all of the products from the *ProductsController*. The method returns a result of type *Pagination* which itself is of type *ProductDto*. The parameters to pass to this controller were way too many so I created a new class called *ProductSpecParams* which includes the maximum page size, the page index, the page size, the brand and category id, as well as the sorting and searching string if any. The [*FromQuery*] annotation specifies that the parameters should be bound using the request query string. The first specification gets a list of all of the products, but also including their types and brands. The second specification returns how many of a particular item is actually available in the entire collection after the filters have been applied. This is needed because it is displayed at the top of the "Shop" page with all of the products. The specification is passed to the *CountAsync*() method and the result is assigned to a variable. The specification is passed to the *ListAsync*() method which returns the products with their types and brands. What follows is a mapper which converts an entity to a data transfer object which will be returned. The DTO shapes the data into a more acceptable format for the client and returns only the properties which are actually needed. In the end, what is returned is the page index which specifies what page the user is on, the page size, the count of the items and the list of items.

```
[HttpGet]
public async Task<ActionResult<Pagination<ProductToReturnDto>>> GetProducts
([FromQuery]ProductSpecParams productParams)
{
var spec = new ProductsWithTypesAndBrandsSpecification(productParams);

var countSpec = new ProductWithFiltersForCountSpecification(productParams);

var totalItems = await _productsRepo.CountAsync(countSpec);
var products = await _productsRepo.ListAsync(spec);

var data = _mapper.Map<IReadOnlyList<Product>,IReadOnlyList<ProductToReturnDto>>(products);

return Ok(new Pagination<ProductToReturnDto>(productParams.PageIndex,
productParams.PageSize, totalItems, data)); }
```

The next piece of code is responsible for displaying the item count next to the basket icon in the navigation bar. It shows the count of products which are currently in the user's basket. In order achieve this, first, in the navigation bar component, I had to inject the basket service and then add a property which is of an observable of type IBasket:

```
this.basket$ = this.basketService.basket$;
```

In the navigation bar's HTML file, I added the following code. The structure (basket$ | async) is an async pipe. It is used to make sure that the observable is disposed of correctly. It subscribes to an observable and returns the latest value it has emitted. It updates continuously which is exactly what is needed so that the correct number appears as the user adds or removes items from the basket.

```
<div *ngIf="(basket$ | async) as basket" class="cart-number"> {{basket.items.length}} </div>
```

Also related to the basket functionality is the ability to increment or decrement the quantity or delete a particular item that is already in the basket. These methods are a part of the basket service class. In the increment method, first the current basket is taken and then it is checked whether there is an item in the basket already with an id the same as the one which is passed to the method. If it is, then the quantity is incremented with one. Then, the basket is updated.

```
 incrementItemQuantity(product: IBasketItem) {
   const basket = this.getCurrentBasketValue();
   const foundItemIndex = basket.items.findIndex((x) => x.id === product.id);
   basket.items[foundItemIndex].quantity++;
   this.setBasket(basket); }
```

The following code fragment is a part of the *AccountsController* and is responsible for registering a user into the system. A Register Data transfer object is passed to the method which includes the information and properties needed for the creation of a new user – username, email and password. First it is checked whether the email in the *registerDto* already exists and if yes, the user is informed. They are not allowed to register. If the email doesn't exist a new instance of *AppUser* is created with the values in the data transfer object. Results is a Boolean variable which returns true or false depending on whether a user has been created or not. When a new user is registering, we are adding them to the "Members" role. Everyone that is registering is a member. At the end, a new *UserDto* is returned with a new token being created because the user is automatically logged in right after they register.

```
[HttpPost("register")]
public async Task<ActionResult<UserDto>> Register(RegisterDto registerDto)
{
if (CheckEmailExistsAsync(registerDto.Email).Result.Value) {
return new BadRequestObjectResult(new ApiValidationErrorResponse{Errors = new []{"Email address is in
use"}});
   }
var user = new AppUser {
        DisplayName = registerDto.DisplayName,
        Email = registerDto.Email,
        UserName = registerDto.Email };
 var result = await _userManager.CreateAsync(user, registerDto.Password);
 if (!result.Succeeded) return BadRequest(new ApiResponse(400));
var roleAddResult = await _userManager.AddToRoleAsync(user, "Member");
if (!roleAddResult.Succeeded) return BadRequest("Failed to add to role");
 return new UserDto  {
               DisplayName = user.DisplayName,
               Token = await _tokenService.CreateToken(user),
               Email = user.Email };  }
```

Below is the *CreateToken*() method from *TokenGenerationService*. It is responsible for creating the token and returning it which will be used in the register and log in methods in the *AccountsController*. In *CreateToken*(), the JWT is built, starting with the claims. Each user has list of claims inside their token. A claim is a piece of information about the user. In my case the claims are the username and email. The role of the user is also added to the claims. The claims will be able to be decoded easily and since the token is stored in the browser, sensitive information should not be put here. Then the credentials are created. This takes the key which was created in the constructor and the algorithm that will be used to encrypt the key. Then what I want inside the token is described. This is the information for the token's payload. First the claims are added, then the token is given an expiry date which is 7 days. The credentials are added and also the issuer of the token. The token is only valid if it was issued by this particular server. The token then needs to be "handled" so a token handler is declared. Then, the token is created using the token descriptor.

```
public async Task<string> CreateToken(AppUser user) {
   var claims = new List<Claim> {
        new Claim(JwtRegisteredClaimNames.Email, user.Email),
        new Claim(JwtRegisteredClaimNames.GivenName, user.DisplayName) };
var roles = await _userManager.GetRolesAsync(user);
claims.AddRange(roles.Select(role => new Claim(ClaimTypes.Role, role)));

var credentials = new SigningCredentials(_key, SecurityAlgorithms.HmacSha256Signature);

var tokenDescriptor = new SecurityTokenDescriptor {
               Subject = new ClaimsIdentity(claims),
               Expires = DateTime.Now.AddDays(7),
               SigningCredentials = credentials,
               Issuer = _config["Token:Issuer"]  };
var TokenHadler = new JwtSecurityTokenHandler();
var token = TokenHadler.CreateToken(tokenDescriptor);
return TokenHadler.WriteToken(token);  }  }
```

5. Testing

Testing is a very important part of the software development cycle. It is a method used in order to check whether the software product meets the expected requirements and to ensure that it doesn't have any defects or "bugs" that disrupt the smooth user experience. Reliable, secure and high performing software is very highly valued. Testing can be done using manual or automated tools. The main goal of the testing process is to discover errors that exist but are not easy to spot for the developer and also to prevent from missing or not fully implementing a particular requirement.

There are different types of tests and not all testing types are applicable to all projects but depend on its nature & scope. For my project, I have conducted acceptance tests the results of which, I will present below. The way the tests were done is by executing the program with both wrong and accurate data and seeing how the software performs. The expected result when wrong data is passed is the software to produce the desired error messages in order to notify the user that something is wrong. When correct data is passed the software is expected to stick to the project's objectives and fulfil the functional and non-functional requirements. In the pages below I will analyze the program's behavior under different circumstances in order to evaluate whether the functional and non-functional requirements have been met.

Before I start, I want to say that no software is perfect, and that no developer can possibly think of all the things that can go wrong and protect against them. This is why projects undergo software quality assurance which is a practice that monitors the software engineering process and methods in order to assure the quality of the software and the conformance to certain standards. The process is extensive and very time-consuming especially in a project that has so many functional requirements as mine. I have attempted to address all of the errors I could think of and prevent the program from crushing as much as possible but there are most probably some bugs that I haven't thought of that are still present in the application.

The fact that there are two components - backend and frontend means that there could be errors related to both which need to be addressed. Some of them are easy to spot by using the Chrome Developer Tools which is a very useful utility since all errors present on the page are displayed in the console. Using Postman and seeing the responses from the API endpoints is also of great help, because it can be determined whether the problem is caused by the backend or it should be looked for somewhere in the frontend.

5.1 Acceptance tests

I will start with presenting the results for the acceptance tests. Acceptance test is a description of the behavior of a software product performed as usage scenario. It is conducted in order to access whether the software meets the expected requirements and specifications. The result of an acceptance test is either pass or fail. They are called acceptance tests because usually through such tests, the client or another entity determines whether to accept the feature or not. There are acceptance criteria that need to be met. Acceptance tests usually make use of the black box testing method and are carried out manually. When it comes to the software development life cycle, acceptance testing is the last level of software testing which is done right before the system becomes available for use and production. The best way to perform these tests is to go through the requirements one by one and see whether the software meets these requirements. Usually in acceptance testing, the requirement might have to be split into a list of smaller features and each of these features to be tested separately.

This is what I have for some of the requirements which have a larger scope. The test whether the software satisfies particular requirement includes how the software performs when invalid data is passed to it. This section also includes the implemented error-handling and how the software performs when the user has made an error.

- **Functional Requirement #1: Any customer should be able to register**

The feature under test here is registration in the system. The first test is performed when the user uses valid data.

- Data used: Valid name, email and password that conform to the rules
- Expected outcome: User is successfully registered in the system and can now use the email and password to log in. User is automatically logged in and redirected to the shop page.
- Test result: This is how the application performs therefore the test is passed.

In the following scenarios, the user enters invalid data:

- Data used: Fields for name and/or email and/or password have been left empty
- Expected outcome: User is not registered. An error message appears right below the text input field indicating that the field is required. The text fields are outlined in red.
- Test result: This is how the application performs therefore the test is passed.

- Data used: An email which doesn't have the format of an email (@ or "." are missing or the word after the "." Is less than 2 characters long).
- Expected outcome: User is not registered. An error message appears right below the text input field indicating that the email address is invalid. The text field is outlined in red.
- Test result: This is how the application performs therefore the test is passed.

- Data used: An email which has already been used for registration
- Expected outcome: User is not registered. An error message appears below the email input field saying that the email has already been used. Text field is outlined in red.
- Test result: This is how the application performs therefore the test is passed.

- Data used: A password which is not long or complicated enough and does not conform to the password rules.
- Expected outcome: User is not registered. An error message appears indicating that the password must have at least one uppercase, one lowercase, one non-alphanumeric character, one number and should be at least 6 characters long. Text field is outlined in red.
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #2: Any customer should be able to log in**

The feature under test here is logging in the system. The first test is performed when the user enters valid data.

- Data used: Valid email and password with which the user has registered before.

- Expected outcome: User is successfully logged in the system. They can see their username in the navigation bar. A token has been generated for them. They can now navigate to the checkout page.
- Test result: This is how the application performs therefore the test is passed.

In the following scenarios, the user uses invalid data:

- Data used: No email and/or password have been entered in the text fields
- Expected outcome: User is not logged in. An error message appears right below the text input field indicating that the field is required. “Login” button is disabled.
- Test result: This is how the application performs therefore the test is passed.

- Data used: An email which doesn’t have the format of an email (“@” or “.” are missing, the word after the “.” Is less than 2 characters long).
- Expected outcome: User is not logged in. An error message appears right below the text input field indicating that the email address is invalid. The text field is highlighted in red.
- Test result: This is how the application performs therefore the test is passed.

- Data used: An email and password combination which has not been registered in the system before
- Expected outcome: User is not logged in. A notification appears indicating that the user is not authorized.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #3: Any customer should be able to browse the catalogue of items</u>**

The feature under test here is seeing all of the items offered by the store.

- Data used: The user is located on the “Shop” page. They are either logged-in or not.
- Expected outcome: User sees all items offered by the store arranged in a grid with their image, name, price and two buttons. One is used for adding the item to the cart and the other for showing details about the item.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #4: Any customer should be able to view detailed information about an item</u>**

Each item has page which offers detailed information about it. No authentication is required for seeing that page. The user doesn’t pass any data to the system. No errors should occur. The feature under test is seeing a page with details about an item.

- Data used: The user has clicked the “Details” button next to a product in the “Shop” page.
- Expected outcome: User is redirected to a page which shows the product’s image, a mini gallery with an option to click on an image, the product’s name, price, description, two buttons for changing the quantity and an “Add to cart” button.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #5: Any customer should be able to filter the items by category or brand</u>**

On the “Shop” page, there should be two tables. One contains all of the brands and the other all of categories offered by the store. The user can click on their names.

The feature under test here is showing products only from a particular brand

- Data used: The user clicks on one of the brands in the “Brands” table on the “Shop” page.
- Expected outcome: The main part of the screen which shows all products is now updated and displays items only from one particular brand
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is showing the products only from a particular category

- Data used: The user clicks on one of the categories available in the “Categories” table on the “Shop” page.
- Expected outcome: The main part of the screen which shows all products is now updated and displays items only from one particular category (for example nuts, flour, chutney, grains, etc.).
- Test result: This is how the application performs therefore the test is passed.

The third feature under test is showing products only from a particular category and brand

- Data used: The user clicks on one of the available brands and one of the available categories.
- Expected outcome: The main part of the screen, which shows all products, is now updated and displays items only from the chosen brand which also match the chosen category
- Test result: This is how the application performs therefore the test is passed.

The last feature under test is removing the filters and show all of the items again

- Data used: User clicks on the “All” option in the Brands and/or Categories table
- Expected outcome: The filters are removed, and all products are displayed regardless of their brand and/or category.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #6: Any customer should be able sort the items by name and price (ascending and descending)</u>**

The “Sort” menu is displayed on the “Shop” page above the table with all items. It displays the three sorting options – Alphabetical, Price: Low to High, Price: High to Low. Authorization is not required. The user is not inputting data and no errors should occur.

The first feature under test here is sorting the products alphabetically

- Data used: User chooses “Alphabetical” from the Sort dropdown menu
- Expected outcome: The products available are sorted by their name. The items whose name begins with a number are shown first. If the name starts with a letter, the standard order of the letters in the alphabet is followed. This is the default

sorting, and the user sees a change only if some other sorting has been applied previously
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is sorting the products with the cheapest appearing first
- Data used: The user clicks on the “Price: Low to High” option from the Sort menu
- Expected outcome: The products are rearranged and the ones with the lowest price appear first. The ordering continues according to the price.
- Test result: This is how the application performs therefore the test is passed

The third feature under test is sorting the products with the most expensive ones appearing first
- Data used: The user clicks on the “Price: High to Low” option from the Sort menu
- Expected outcome: The products are rearranged and the ones with the highest price appear first. The ordering continues according to the price.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #7: Any customer should be able to search for an item</u>**

The “Search” text box and buttons are located on top of the page with all items.

The first feature under test here is searching for an item
- Data used: User writes a word or several words in the text box. Then they press the “Search” button.
- Expected outcome: The products whose name contains the search word or words appear on the page. All other products disappear. If any filter or sorting has been chosen, it still applies.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is seeing all items again after a search
- Data used: The user clicks the “Reset” button, which is located next to the “Search” button
- Expected outcome: All of the items appear again
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #8: Any customer should be able to add an items to their shopping cart</u>**

Buttons with a shopping cart icon exist for each of the items. The buttons can be found next to an item on the “Shop” page and also in each of the item’s Details page.

The first feature under test here is **adding an item to the shopping cart**
- Data used: User clicks on one of the buttons for adding an item to the shopping cart
- Expected outcome: The item is added to the user’s shopping cart in the database. The number next to the shopping cart icon in the navigation bar is incremented. The newly added item can now be seen in the “Shopping cart” page
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is the user seeing the items which they have added to their cart so far

- Data used: The user clicks on the shopping cart button in the navigation bar.
- Expected outcome: The user is redirected to the page where they can see all items, they have added to their cart so far. The items appear in table with several columns (name, price, quantity, total and remove).
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #9: The information in the shopping cart should be persisted for further sessions of the particular user**

When the user adds something to their basket for the first time, a basket is created, and its Id is stored in the local storage of the browser. So, the contents of the basket are retrieved when the user visits the page again. In this way they don't have to add the items each time they visit the page. The test is passed.

- **Functional requirement #10: Any customer should be able to modify the contents of their shopping cart**

In the shopping cart page, users are able to increase or decrease an item's quantity. They also have the option to remove an item from their carts

The first feature under test here is **increasing the quantity of an item**

- Data used: User clicks on the "+" button next to the quantity number
- Expected outcome: The item's quantity is increased. The total price for an individual item is increased accordingly. The total price for all items is increased accordingly.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **decreasing the quantity of an item**

- Data used: User clicks on the "-" button next to the quantity number
- Expected outcome: The item's quantity is decreased. If the quantity was 1, the item is removed from the cart. The total price for an individual item decreases accordingly. The total price for all items is decreases accordingly.
- Test result: This is how the application performs therefore the test is passed.

The third feature under test is **removing an item from the shopping cart.**

- Data used: User clicks the trash bin icon next to a product.
- Expected outcome: The product is removed from the shopping cart. It is no longer visible in the table. The number next to the shopping cart icon in the navigation bar is decreased.
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #11: Any customer should be able to see the total cost of their purchases in the shopping cart page**

The first feature under test here is **seeing the total cost for an individual item**

- Data used: User has added items to their cart with a quantity of either one or more

- Expected outcome: Users sees the total price for an individual product which is calculated as price for one multiplied by item quantity
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **seeing the total cost for all items in the cart**
- Data used: User has added items to their cart
- Expected outcome: Users sees the total price for all products which is calculated by summing the total price for each of the individual product
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #12: When making an order, authenticated customers should be able to input their delivery address</u>**

This is one of the steps of the checkout process which the user needs to complete in order to place an order. It is a form with text fields which the users have to fill. Only authenticated customers should be able to get to this stage.

The first feature under test here is **accessing the Checkout page and seeing the delivery method form**
- Data used: User clicks on the “Proceed to Checkout” button in the basket page and user is logged in their account.
- Expected outcome: User is redirected to the Checkout stepper component with the first step being inputting the address
- Test result: This is how the application performs therefore the test is passed.

In the second scenario of this feature test, the user is not logged in
- Data used: User clicks on the “Proceed to Checkout” button in the basket page and user is not logged in their account.
- Expected outcome: The user is redirect to the “Login” page.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **address form being populated automatically**
- Data used: The user has already made an order in the system or saved an address before
- Expected outcome: The fields in the delivery address form are automatically filled
- Test result: This is how the application performs therefore the test is passed

The third feature under test is **the user updating their default address in the system**
- Data used: User clicks on the “Save as default address” button, located on top of the Address form
- Expected outcome: The address is now saved for future orders and will be inserted into the form automatically when the user makes future purchases
- Test result: This is how the application performs therefore the test is passed.

The last feature under test is **filling in the delivery address information.** In the first scenario the user enters valid data.
- Data used: The user fills all of the required fields – first name, last name, street, city, country and postcode.

- Expected outcome: The text fields are outlined in green. The address is accepted and will be saved in the database for the particular order. The user is allowed to continue to the next step of the checkout process.
- Test result: This is how the application performs therefore the test is passed.

In the second scenario the user uses invalid data.

- Data used: The user doesn't fill in all of the fields in the address form and leaves some of them empty.
- Expected outcome: An error message appears below the empty text box indicating that the field is required. The text box is outlined in red. User is not allowed to continue to the next step of the Checkout process before they have resolved all errors. The button for the next step is disabled.
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #13: When making an order, authenticated customers should be able to choose one of several shipping options**

The first feature under test here is **seeing the shipping options**

- Data used: User clicks "Go to Shipping" button
- Expected outcome: Users sees a list of the available shipping options which are overnight shipping, express ground shipping, regular ground shipping and free. There is an empty square box next to each of them.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **choosing a shipping option**

- Data used: User clicks on one of the square boxes next a shipping method's name
- Expected outcome: The shipping method for the order is saved. The total price for the order increases and now includes the shipping price as well, along with the price of the items. The user is allowed to go to the next step of the Checkout process which is Review.
- Test result: This is how the application performs therefore the test is passed.

There is also a second scenario related to the feature above

- Data used: The user doesn't choose any of the shipping options.
- Expected outcome: The "Go to Review" button is disabled, and the user is not allowed to go onto the next step of the Checkout process
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #14: When making an order, authenticated customers should be able to see their order summary information**

The feature under test here is **reviewing the order**

- Data used: User clicks "Go to Review" button
- Expected outcome: The user sees a table with all of the products which were in their basket which will be a part of their order, along with their quantity and price
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #15: When making an order, authenticated customers should be able to fill in their credit card information**

The first feature under test here is **seeing the credit card form**

- Data used: User clicks “Go to Payment” button
- Expected outcome: The user sees a form for filling in their credit card information. The form contains text fields for name, card number, expiration date and CVC.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **filling in the credit card form**

- Data used: The user fills in all of the required fields.
- Expected outcome: User is allowed to submit the order. The credit card information will not be validated because the website is not connected to an online payment processing platform
- Test result: This test does not pass because the user’s card information is not verified. This will have to be done by an online payment platform. The only validation is that the user fills in all of the fields. At this point, no money is taken from them.

There is also a second scenario related to the feature above with invalid input

- Data used: The user leaves some or all of the fields in the form empty
- Expected outcome: An error message appears below the empty text box indicating that the field is required. The text box is outline in red. User is not allowed to Submit the order. The button is disabled.
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #16: An authenticated customer should be able to place an order and see an order confirmation page with the order’s details**

The first feature under test here is **placing an order**

- Data used: The user clicks on the “Submit Order” button
- Expected outcome: The order is saved in the database. A toast notification appears saying that the order has been crated successfully. The user is redirected to a page with the text “Thank you. Your Order is confirmed”, along with a “View your order” button.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **seeing the details of the current order**

- Data used: User clicks on the “View your order” button from the confirmation page
- Expected outcome: User sees a table which contains the name of the product, individual price, quantity and total price (individual * quantity) as well as the order total price, including items subtotal and shipping.
- Test result: This is how the application performs therefore the test is passed.

- **Functional requirement #17: An authenticated customer should be able to view details about their previous orders**

The first feature under test here is **seeing the previous orders**

- Data used: The user clicks on the “View Orders” button in the navigation bar

- Expected outcome: The user is redirected to a page with a table which includes their previous orders. The table has columns for order number, date, total and status.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **seeing details about the previous orders**

- Data used: The user clicks on any of the orders in the table from the previous step
- Expected outcome: User sees a table which contains the name of the product, individual price, quantity and total price (individual * quantity) as well as the order total price, including items subtotal and shipping.
- Test result: This is how the application performs therefore the test is passed.

The next four acceptance tests are related to the functional requirements for the administrators.

- **<u>Functional requirement #18: Only authenticated administrators should be able to access the administrator panel</u>**

The feature under test in this requirement is the admin **seeing the administrator panel**. The rule is that the user is in an administrator role.

- Data used: The admin should click on the “Admin” menu link from the navigation bar
- Expected outcome: The outcome should be a table with all of the products available in the store. For each product there is an “Edit” and “Remove button”. There is a “Create new product” button above the table.
- Test result: This is how the application performs therefore the test is passed.

When the rule is not fulfilled and the user is not an administrator, they cannot see the “Admin” navigation link, but they might still try to access the admin endpoint.

- Data used: A user is trying to access the /admin endpoint by typing it in the URL bar.
- Expected outcome: The user will not be allowed to do that because the route is protected by the client. Even if they somehow manage to get to this page, they will not be able to create, edit or delete anything because these endpoints are protected in the API.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #19: An administrator should be able to create a new item</u>**

The first feature under test is **seeing the “Create” item form**. The rule is that the user is in an administrator role.

- Data used: Admin clicks “Create new product” button.
- Expected outcome: Admin is redirected to a page where a new item can be created. There are text fields for product name, price, description and dropdown menus for brand and category which contain the available brand and categories. The images for an item can only be added through editing.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **creating a new item.** The first test is performed when the admin enters valid data.

- Data used: Admin fills all of the fields.
- Expected outcome: A new item is created and added to the inventory. It can now be seen on the “Shop” page. The item’s image is set to a default placeholder image which can be changed via editing the item
- Test result: This is how the application performs therefore the test is passed.

In the second test, the admin enters invalid data.

- Data used: Admin leaves some or all of the fields empty.
- Expected outcome: An error message appears below the empty text box indicating that the field is required. The text box is highlighted in red. The new item is not created. The Submit button is disabled.
- Test result: This is how the application performs therefore the test is passed.

The third feature under test is adding a price for the item.

- Data used: Admin inputs a price which is below 0.
- Expected outcome: An error message appears informing the admin that the price cannot be below 0.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #20: An administrator should be able to edit information about an existing item</u>**

The first feature under test is **seeing the “Edit” item form.** The rule is that the user is in an administrator role.

- Data used: Admin clicks “Edit” button for a particular item.
- Expected outcome: Admin is redirected to a page where the item can be edited. All of the fields are already completed with the details of the item. The user can change their values.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **editing an item**. The first test is performed when the admin enters valid data.

- Data used: Admin doesn’t leave any of the fields empty and the price is above 0
- Expected outcome: The information about the item is updated in all of the views and the database as well.
- Test result: This is how the application performs therefore the test is passed.

In the second test, the admin enters invalid data.

- Data used: Admin leaves some or all of the fields empty or leaves the price at 0
- Expected outcome: An error message appears below the empty text box indicating that the field is required. An error message appears under the price textbox saying that the price cannot be 0. The text box is highlighted in red. The item is not updated. The Submit button is disabled.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #21: An administrator should be able to remove an item</u>**

The feature under test in **removing an item from the inventory**. The rule is that the user is in an administrator role.

- Data used: Admin clicks “Remove” button for a particular item in the table
- Expected outcome: The item is removed from the inventory and is no longer visible in any of the views.
- Test result: This is how the application performs therefore the test is passed.

- **<u>Functional requirement #22: An administrator should be able to upload additional photos, set a main photo for an item and remove existing photos</u>**

The first feature under test is **accessing the “Edit photos” section**. The rule is that the user is in an administrator role.

- Data used: Admin clicks “Edit photos” section in the Edit item form
- Expected outcome: The edit photos section is displayed. It includes the photos that have been uploaded for the item with two buttons below each photo. One is the “Set main” button and the other one has trash can icon displayed. There is a button for adding a new photo at the top of the form.
- Test result: This is how the application performs therefore the test is passed.

The second feature under test is **accessing the page for adding a new photo**

- Data used: Admin clicks on “Add new photo” button in the Edit photos form
- Expected outcome: Admin is redirected to the page for adding a new photo. There is a dropzone for adding an image, a section for cropping the image and a third one for previewing it.
- Test result: This is how the application performs therefore the test is passed.

The third feature under test is **uploading a new photo.** The first test is performed when the admin uses valid data.

- Data used: Admin clicks on the “Drop image” dropzone and selects an image FILE from their computer.
- Expected outcome: The image is uploaded and can be viewed on the page. The admin can crop the image and change its size. As they move the cropper, the image preview changes accordingly. With the “Upload image” button the image is successfully uploaded and added to the gallery of the item which can be found on the item’s Details page.
- Test result: This is how the application performs therefore the test is passed.

In the second part of this test, the admin uses invalid data.

- Data used: Admin chooses a file to upload that does not have .jpg or .png file extension.
- Expected outcome: File is not uploaded. Files which are not images cannot be uploaded. Nothing is displayed in the page and nothing is added in the gallery.
- Test result: This is how the application performs therefore the test is passed.

The fourth feature under test is **setting a main photo**

- Data used: Admin clicks on "Set main" button below one of the images for an item
- Expected outcome: The item's image is updated in all of the other views. For example, in the "Shop" page, the item now appears with its new image.
- Test result: This is how the application performs therefore the test is passed.

The last feature under test is **removing an image.**

- Data used: Admin clicks the button with the trash can icon below one of the images
- Expected outcome: The image is removed from the item's gallery. An item's main image cannot be deleted. First a new main image should be said and then the old one deleted.
- Test result: This is how the application performs therefore the test is passed.

6. <u>Results and conclusion</u>

Over the course of several months, I started from "nothing" and managed to build an e-commerce store using .NET and Angular. I can say that I am pleased with the way the application turned out and that my initial expectations were met. In the beginning, I had a plan to implement at least a few more functionalities than what has currently been implemented but, in the end, I decided that making sure the core functionalities work in the best way possible is more important than having lots of different things, so I concentrated my time and efforts on extending and polishing them rather working on new ones. I am pleased that everything works properly, and no errors are produced in the backend and frontend. I am also happy with the design of the application and the overall look and feel. Another great thing is that things not only look good on the outside, but I have also strived to follow the standards for writing good code and to organize it as much as possible in separate files and folders so that if other people were to work on this application in the future, it would be easier to find what they are looking for. The software architecture guidelines and the design patterns have helped me achieve this. Separating the code into different classes, files and folders and grouping similar functionalities together was a time-consuming process but I believe spending time on it was very much worth it. Even though there are still many things which could be improved and many more functionalities which could be added, the main functionalities of an e-commerce website have been implemented successfully. With a bit more work on validation, performance and security, I believe this website could actually be used for a real online store. Having built the foundation of the application, extending it and accommodating some changes would be a straight-forward process.

I will list some of the things I have managed to achieve during the process of budling this application. Starting with the backend, I have managed to create a multi project .NET Core application while using the dotnet CLI. I have used the Repository and Specification design patterns in .NET. I have created databases and have used Entity Framework and multiple DbContexts. I also set up Redis to store the user's shopping baskets. I have used ASP.NET Identity for login and registration and implemented a token-authentication service. I implemented paging, sorting, searching and filtering. For the frontend, I have created a client-side Angular UI for the store using the Angular CLI. I have used Angular modules to create lazy loaded routes, Bootstrap to create and design the User Interface, and many more Angular technologies, such as form components, observables, auth, the async pipe, etc.

I faced numerous challenges during development. There were some really stubborn bugs which I spent days resolving. Luckily, both .NET and Angular are well-established and widely used frameworks so there are a lot of resources on the Internet which could be referenced when working with them. They are also very often paired together. Both frameworks also come with many services and features out of the box that developers do not need to worry about. I find it very useful that both frameworks offer clear guidelines and suggestions for developers on how to do things properly in their broad and detailed documentations. The large community of developers on Stack Overflow is also very helpful. While I was familiar with some of concepts and have implemented some of the functionalities before, either on past university projects or during my web development internship, for others, I had to spend a significant amount of time reading and researching before actually implementing the feature and writing the code. Such was the case with the token-authentication system, which was one of the most challenging features to implement. However, knowing the importance of security in the current era of web services, I believe that being familiar with authentication and authorization and knowing how to implement them is a highly valued skill for a developer.

As much as I wanted to include as many things as possible, the limited time scope of this project didn’t allow me to. However, I have decided to keep working on it and keep extending it because I believe it will help me become a better web developer and will be great project to add for my portfolio when applying for jobs. There is plenty of opportunity to extend this application. The first thing I would add is a facility to accept payments from the user and to verify the credit card information that they input. For this project, I didn’t have the time and resources to do it especially because it requires working with third party services which very often require some kind of a subscription fee. Nevertheless, if the website was to work with real customers, this step would be necessary. I would also extend the inventory system and add a number which indicates how many of the products are in stock. This number would then go down depending on the orders users make. I would also implement an email service. I even looked into SendGrid which is such an email service that can be integrated with .NET. Having an email service would give me the opportunity to verify user’s email address and implement a password changing system and do much more things which are necessary for a website nowadays. Another thing which I think has become a standard for e-commerce websites is a product review system. The users would be able to rate products and leave comments about them. Customer feedback is an extremely important tool for businesses and incorporating it in the e-commerce solution would drive conversation and help build trust and increase sales. I also looked into making a recommendation system which would recommend similar products to what the users have bought recently or what they have looked at or at least it would recommend similar products to the one the user is looking at right now.

I would say that although challenging, the work done on this project has been a great learning experience. I have learned a plethora of new things and web development techniques which have made me a more confident web developer and a better problem solver. I have also learned how to manage my time better when working on a large-scale project by myself with a clearly defined deadline. The project will be added to my portfolio and I am sure I will mention it to my future employers when I start looking for a job.

For screenshots of the application running, please look at the "Description of the user interface" part of this thesis.

7. References

[1] https://www.clariontech.com/blog/cloud-computing-architecture-what-is-front-end-and-back-end

[2] https://spin.atomicobject.com/2015/04/06/web-app-client-side-server-side/

[3] https://www.forbes.com/sites/forbestechcouncil/2018/07/19/seven-reasons-why-a-websites-front-end-and-back-end-should-be-kept-separate/?sh=75ff3fde4fca

[4] https://docs.microsoft.com/en-us/dotnet/architecture/modern-web-apps-azure/common-web-application-architectures

[5] https://jeffreypalermo.com/2008/07/the-onion-architecture-part-1/

[6] https://blogs.u2u.be/peter/post/building-a-generic-repository-using-the-specification-pattern

[7] https://angular.io/guide/architecture

[8] https://angular.io/guide/template-syntax

[9] https://www.computer.org/publications/tech-news/trends/10-essential-steps-to-improve-your-website-security

[10] https://andrewlock.net/exploring-the-asp-net-core-identity-passwordhasher/

[11] https://jwt.io/introduction

[12] https://dotnet.microsoft.com/apps/aspnet

[13] https://www.typescriptlang.org

[14] https://www.altexsoft.com/blog/engineering/the-good-and-the-bad-of-angular-development/

[15] https://angular.io/guide/what-is-angular

[16] https://www.toptal.com/front-end/what-is-bootstrap-a-short-tutorial-on-the-what-why-and-how

[17] https://dotnettutorials.net/lesson/generic-repository-pattern-csharp-mvc/