National College of Ireland

Star Nutrition

BSHCSDE

Academic Year i.e. 2022/2023

Colin Shaw

X19145586

X19145586@student.ncirl.ie

Star Nutrition

Technical Report

Contents

Executive Summary 1

1.0 Introduction 1

1.1. Background 1

1.2. Aims 2

1.3. Technology 2

1.4. Structure 3

2.0 System 3

2.1. Requirements 3

2.1.1. Functional Requirements 4
2.1.1.1. Use Case Diagram 4
2.1.1.2. Requirements: 4
2.1.1.3. Description & Priority 5
2.1.1.4. Use Case 7
2.1.2. Data Requirements 12
2.1.3. User Requirements 12
2.1.4. Environmental Requirements 13
2.1.5. Usability Requirements 13
2.2. Design & Architecture 14
2.3. Implementation 106
2.4. Graphical User Interface (GUI) 129
2.5. Testing 146
2.6. Evaluation 165
3.0 Conclusions 165
4.0 Further Development or Research 166
5.0Appendices 167
4.1. Poster 167
4.2. Project Proposal 167

Executive Summary

The report's goal is to give a breakdown of the essential elements and features of Star Nutrition. The reports delve into the application's structure, covering models like Product, Category, Order, and UserProfiles that specify the data architecture and interactions. The operations for the homepage, product listing, and checkout are described, including information on how views manage incoming requests and produce suitable HTTP replies.

The Category and Product models, which classify and represent products with their qualities, are two models that are highlighted in the paper. Order information and line items are managed via the Order and OrderLineItem models, respectively. Additional user data is kept in user profiles. It is explained how views like "products_view" control the organisation, sorting, and searching of product displays. The checkout procedure is divided into parts, starting with form submissions and ending with payment processing via the Stripe API.

The report provides a thorough overview of the fundamental components of the programme, including models, views, and forms. It emphasises how these elements work together invisibly to make Star Nutrition. The report is a reference for readers and developers interested in learning about the internal workings of the Django-based e-commerce platform.

1.0 Introduction

1.1. Background

The goal for undertaking this project is to gain more understanding of full stack development. The idea behind this project is to design an online e-commerce web application. The project should be

easy to use, secure, cleanly coded and not have as much clutter as other applications. I have the intention to get a job as a full-stack developer so undertaking a project like this will be of most benefit to me. I have an interest in sports and nutrition and set up an online business selling supplements.

The main reason I have chosen to undertake this project is to build a full-stack application, as I want to become a full-stack developer. Python is a language I like coding in and this will be the greatest challenge for me and the most rewarding to design, develop and deploy.

To set out and meet my objectives I am going to be very strict with time and deadlines while designing, developing and implementing all code and documentation.

As stated the SMART goals along with using Agile methodology and tracking through Trello will help me accomplish a good quality project along with seeing what is needed to do this project. When starting to design the project I will be real with what needs to be done and overstretch myself and spend more time than I have in previous project planning and leave no minor detail left out.

When developing the application user stories will be used to ensure that I am sticking to how the application should work and that all functionality is getting added correctly and relevantly.

The documentation will be the hardest part, I will have to look at the layout of the previous year's documentation to ensure I am laying out what is needed and implementing it correctly.

I will use the help of my mentor Vladimir as much as possible as this is new to me and his knowledge will be invaluable.

1.2. Aims

The goal of this project is to build a full-stack application from design all the way through to deployment. This will be done using an Agile methodology for project management. I will set out SMART goals, that are specific, measurable, achievable, and relevant. The application will be a nutrition platform that will sell health supplements. The application will be developed using Django a Python framework. it will be developed in GitHub and Gitpod, the reason for this is the ease of use while developing and committing branches to GitHub for tracking the development. It will use Stripe's API for taking secure payments. The application will be deployed using Heroku a cloud-based application. AWS will be used for loading and rendering static files. The main objective is to make a nutrition platform that is easy to use, navigates well, looks well and sets itself apart from previous nutrition applications.

The objective as well as building an application is to write good quality documentation, as if I can document how I have made and designed this application I will have truly done a good project and learnt what I need to out of this project.

1.3. Technology

In my approach to development, I will use user stories for development and form the key functionality of the project. The user stories will be added to the software requirements and then implemented throughout the design and development phases, if they all work then the application will work.

An agile methodology will be used throughout the whole project in design, development and documentation. It will be tracked using Trello and with my own plan and the help of my supervisor, it will be done with an agile approach.

GitHub has a new user stories section so I also plan to use this in implementing the user stories and the agile approach as it will benefit my documentation of the project.

The software requirement section will identify the main requirements of the functionality of the project and help make a blueprint of what's needed. More on this in section 5 the technical details section below.

During the design phase, I plan to implement core functionality, user experience, quality products, and good database designs and models to ensure the applications is of great quality.

1.4. Structure

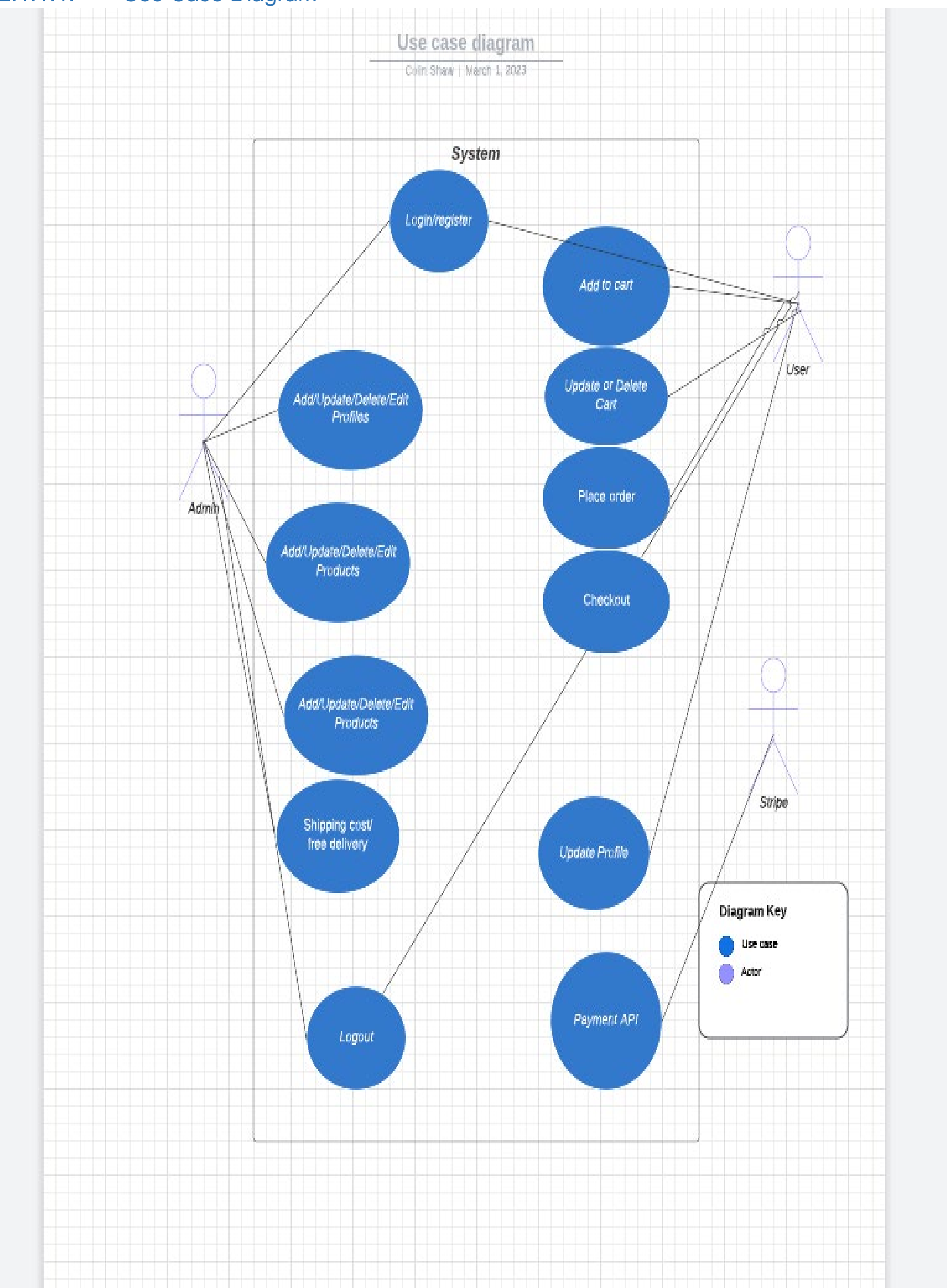
Provide a brief overview of the structure of the document and what is addressed in each section.

2.0 System

2.1. Requirements

2.1.1. Functional Requirements

2.1.1.1. Use Case Diagram



2.1.1.2. Requirements:

1. User Requirement 1: User registration
2. User Requirement 2: User Login
3. User Requirement 3: Add product to cart.
4. User Requirement 4: Update or remove items from the cart.
5. User Requirement 5: Place order

6. User Requirement 6: Checkout
7. User Requirement 7: Update user profile
8. User Requirement 8: Logout

1. Stripe Requirement 1: Payment

1 Admin Requirement 1: Log in
2 Admin Requirement 2: Add Products
3 Admin Requirement 3: Edit Products
4 Admin Requirement 4: Delete Products
5 Admin Requirement 5: Calculate Shipping costs.

2.1.1.3. Description & Priority

User Requirement 1:

The user registration user case will describe how a user registers for Star Nutrition's system. Once on the application, a user can click register and be directed to the form page where the user will fill out their details like username, password, email and address. This is a high-priority requirement as a user has to be able to have a profile to be able to checkout.

User Requirement 2:

The user can log in once they have registered for an account. Once on the applications a user can click login and enter their email and password to log in. then they will be brought to the logged-in home screen. This is a high-priority requirement as users must have an account to update their profile and buy products.

User Requirement 3:

The user must be able to add a product or a certain quantity of product to their shopping cart. This is an essential product requirement as registered users will have to be able to select the products they want to buy.

User Requirement 4:

The user must be able to remove products or a certain quantity of products from their shopping bag. This is an essential product requirement as registered users will have to be able to remove items if they do not want to purchase products.

User Requirement 5:

The user must be able to place an order. When a user has selected the products they want to buy, they must then be able to place an order to buy the product. This is a high-priority requirement as orders need to be placed and details entered correctly.

User Requirement 6:

A user must be able to check out when they place an order and confirm details. Once a user has entered their details they must be able to check out and buy their selection. This is a high-priority requirement.

User Requirement 7:

The user must be able to update their details. The user must be able to change addresses, passwords and other information. This is a high-priority requirement as users

must be able to change passwords or update details easily to ensure the application works well.

User Requirement 8:

The user must be able to log out when their session is over. The user needs to click log out and they must be logged out and the session ends immediately. This is a high-priority requirement as users need to be able to log out after each session.

Stripe Requirement 1:

A user's payment details must be kept safe and free of any security breaches. Stripes API will help manage online payments securely. This is a high-priority requirement.

Admin Requirement 1:

An Admin user must be able to log in. Admin users will be super users and able to log in to edit products. This is a high-priority requirement to ensure the application can be kept up to date.

Admin Requirement 2:

An Admin user must be able to Add/Update products. When a new product launches a user must be able to add this easily. This is a high-priority requirement.

Admin Requirement 3:

An Admin user must be able to edit products. When a product price or details need to be updated the admin user will be able to edit these details easily. This is a high-priority requirement.

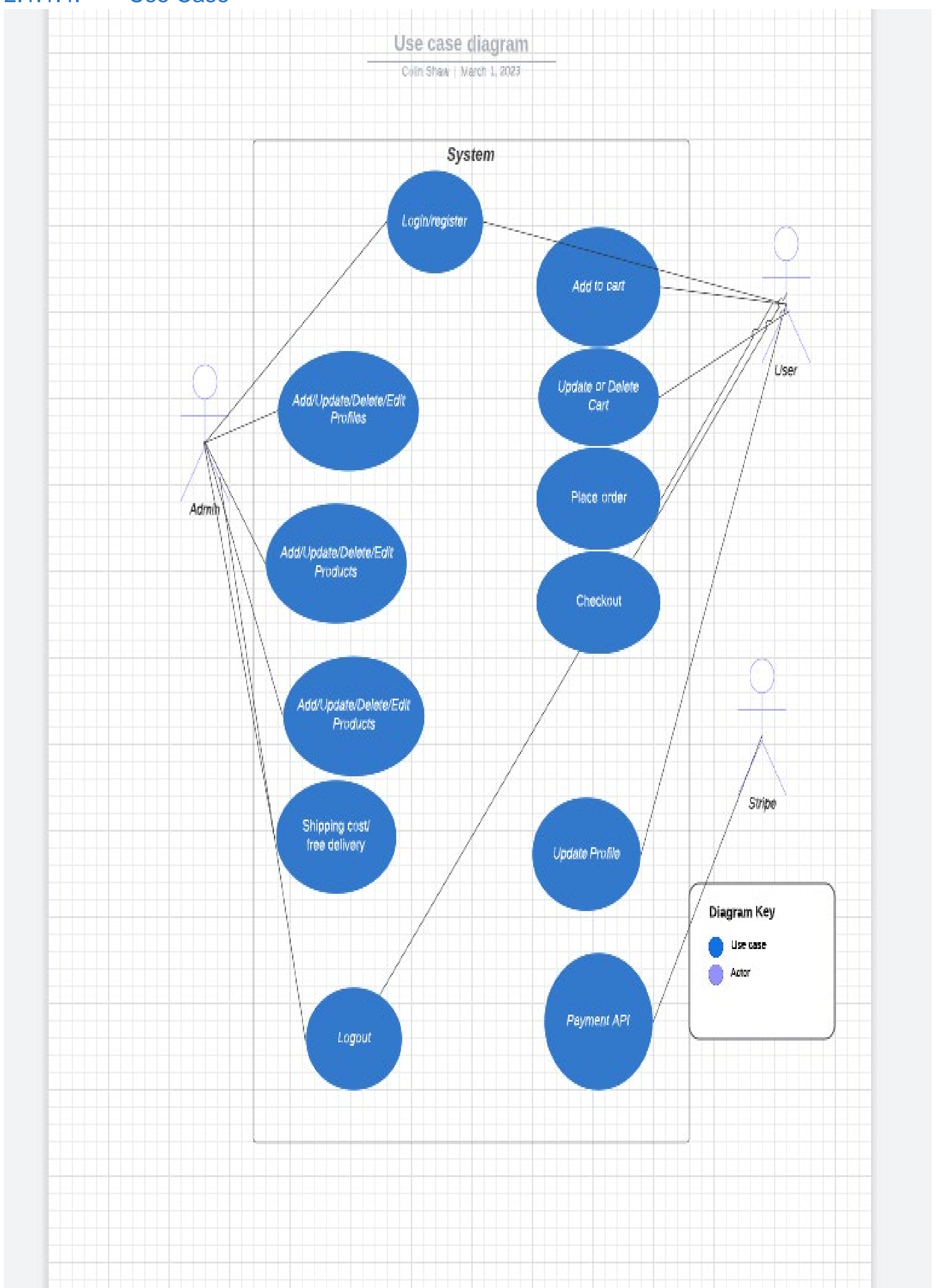
Admin Requirement 4:

An Admin user must be able to delete products. When a product is discontinued the admin user must be able to take it off the site. This is a high-priority requirement.

Admin Requirement 5:

And admin users must be able to calcite shipping costs depending on delivery distance or free delivery over a certain price. This is a high-priority requirement.

2.1.1.4. Use Case



`

User Requirement 1: User registration

Scope

The scope of this use case is for a potential customer who wants to register for an account.

Description

The user registration user case will describe how a user registers for Star Nutrition's system. Once on the application, a user can click register and be directed to the form page where the user will fill out their details like username, password, email and address. This is a high-priority requirement as a user has to be able to have a profile to be able to checkout.

Flow Description

Precondition

The user has no registered account on the Star Nutrition application.

Activation

This use case starts when the user clicks to register.

Main flow

1. The user navigates to the register page form.
2. The user enters their details.
3. The User must enter their name, email address, address, number and password.
4. The system checks details are okay.
5. The system validates the new account.
6. The user will be sent a verification email.
7. The user accepts the email verification.
8. The user registers for an account.

Alternate flow

A1: The password does not meet the requirements.

1. The user navigates to the register page form.
2. The user enters their details.
3. The User must enter their name, email address, address, number and password.
4. The system checks details are okay.
5. The system does not accept the password as no special characters are entered.
6. The user enters their details with the correct password.
7. The system checks details are okay.
8. The system validates the new account.
9. The user will be sent a verification email.
10. The user accepts the email verification.
11. The user registers for an account.

Exceptional flow

E1: Users' email already in use.

1. The user navigates to the register page form.
2. The user enters their details.
3. The User must enter their name, email address, address, number and password.
4. The system checks details are okay.
5. The system does not accept email as it is in use for another account.
6. The user finds an old account and logs in.

Termination

The system presents the logged-in home page and users can now start shopping or log out.

Postcondition

The system goes into a wait state.

User Requirement 2: User Login

Scope

The scope of this use case is for a customer who wants to log into the application.

Description

The user can log in once they have registered for an account. Once on the applications a user can click login and enter their email and password to log in. then they will be brought to the logged-in home screen. This is a high-priority requirement as users must have an account to update their profile and buy products.

Flow Description

Precondition

The user has a registered account on the Star Nutrition application.

Activation

This use case starts when the user clicks login.

Main flow

1. The user navigates to the home page.
2. The user clicks login.
3. The User must enter their username and password.
4. The system checks details are okay.
5. The system validates and lets the user log in.
6. The user is redirected to the homepage.
7. The user can start shopping.

Alternate flow

A1: The user needs to reset the password.

1. The user navigates to the home page.
2. The user clicks login.
3. The User must enter their username and password.
4. The system checks details are okay.
5. The system does not allow passwords.
6. The user has forgotten the password.
7. The user clicks forgotten password.
8. The user is directed to the forgotten password page.
9. The user enters their email.
10. The system sends an email verification link to reset the password.
11. The user clicks the email link to reset the password.
12. The user resets the new password.
13. The user tries to log in.
14. The system checks details are okay.
15. The system validates and lets the user log in.
16. The user is redirected to the homepage.
17. The user can start shopping.

Exceptional flow

E1: The user cannot gain access due to network issues.

1. The user has no internet connection.
2. The user cannot gain access to the site.

Termination

The system presents the logged-in home page and users can now start shopping or log out.

Postcondition

The system goes into a wait state.

User Requirement 3: Add product to cart.

Scope

The scope of this use case is for a customer who wants to select a product.

Description

The user must be able to add a product or a certain quantity of product to their shopping cart. This is an essential product requirement as registered users will have to be able to select the products they want to buy. **Use Case Diagram**

Flow Description

Precondition

The user is logged in and on the home page.

Activation

This use case starts when a user selects a product.

Main flow

1. The user is browsing the category of products.
2. The user selects a product.
3. The user is brought to the product details page.
4. The user chooses a quantity of 2 of the products.
5. The user then adds the products to the cart.
6. The system adds the products to the cart.
7. The system asks does the user want to continue shopping or go to the cart.
8. The user selects keep shopping.

Alternate flow

A1: Item already in the user's shopping cart.

1. The user is browsing the category of products.
2. The user selects a product.
3. The user is brought to the product details page.
4. The user chooses a quantity of a single product.
5. The system checks and the item are already in the user's cart.
6. The system shows a message to the user that this item is already in their cart and that would they like to add this product again.
7. The user selects yes add to a cart.
8. The system asks does the user want to continue shopping or go to the cart.
9. The user selects keep shopping.

Exceptional flow

E1: The user does not have a registered account.

1. The user is browsing the category of products.

2. The user selects a product.
3. The user is brought to the product details page.
4. The user chooses a quantity of a single product.
5. The system checks if a user is registered and sees they are not.
6. The system asks the user if they would like to set up an account.
7. The user chooses yes to set up an account or not.

Termination

The system lets the user continue shopping or brings them to the shopping cart.

Postcondition

The system goes into a wait state.

Admin Requirement 2: Add Products

Scope

The scope of this use case is an admin user who wants to add a product.

Description

An Admin user must be able to Add/Update products. When a new product launches a user must be able to add this easily. This is a high-priority requirement.

Flow Description

Precondition

The admin user is logged in and at the home page.

Activation

This use case starts when an admin user selects add products.

Main flow

1. The admin user selects add product.
2. The admin user is brought to the add product page form.
3. The admin enters the product details and image.
4. The system checks product details.
5. The system confirms users want to add products.
6. The admin user selects add product.
7. The product is added.
8. The system asks would the user like to add another product.

Alternate flow

A1: Product already added.

1. The admin user selects add product.
2. The admin user is brought to the add product page form.
3. The admin enters the product details and image.
4. The system checks product details.
5. The system confirms this product is already and asks the user do they still want to add this product.
6. The admin users select no and choose to add another product.

Exceptional flow

E1: Technical issues.

1. The user encounters technical issues and cannot access the application.

Termination

The system lets the user add the product.

Postcondition

The system goes into a wait state.

2.1.2. Data Requirements

- **User Information:**
  This will include the user's personal information such as name, email, password, number, address, payment details and order history. Users will have customisable profiles for updating their details.
- **Admin Information:**
  This will include the super user's information such as name, email, passwords and permissions. Admins will be able to add, delete or edit products.
- **Products:**
  This will include the product details such as product id, images, category, description and price.
- **Comments and reviews:**
  This will hold the details of the comments and reviews of the product. To help improve the applications.
- **Applications data:**
  The application will capture data such as usage, performance stats, and webpage views. This will help improve the application.
- **Security and data:**
  This is how the application will securely hold sensitive data. Encryption to ensure no data breaches.

2.1.3. User Requirements

- **Easy set-up for new users:**
  Users will be able to set up and register an account easily. Name, email, password, number, and address will be needed at registration.
- **Easy login for user:**
  Users will easily be able to log in to the application once registered with their email/username and password.
- **Password Reset:**
  A user will easily be able to reset or update their password details. If a user has forgotten a password it will be no issue to reset their old password with a few easy security steps.
- **Super users:**
  Super users will be set up by authorised personnel. Only superusers/admins can add or update products.
- **Create/Read/Update/Delete(CRUD) easily:**
  An admin user will easily be able to add new products, update them and delete them if necessary once they have the correct permissions.
- **Easily shop products:**
  Users will be able to search for products via categories, names, descriptions and prices. They will also be able to filter products. The products will be easy to navigate

and once a product is clicked they will be able to see all data for the product very clearly.

- **Competitive pricing:**
  The products will be competitively priced compared to the market. Trends will be analysed.
- **Easy checkout:**
  Users will easily be able to check out once they are registered. Once the details entered are correct the user will be able to check out with the click of a button.
- **Easy Payment:**
  The application will use stripes powerful payment system which will add security and put the users at ease buying products.
- **Easily leave product reviews:**
  Users will be able to easily review products on a scale of 1-5 stars and leave a comment about their experience.

2.1.4.Environmental Requirements

- **Application performance:**
  The application will need a device with a stable internet connection to function on the internet.
- **Multiple devices:**
  Any device mobile or tablet, or internet explorer chrome will be able to render the application.
- **SEO:**
  Full use of Search Engine Optimisation will be used to maximise customer reach along with marketing.
- **Compliance with industry standards:**
  The application will comply with all laws and requirements so it is fully legitimate.
- **Database performance:**
  The application will have sufficient capacity for performance online. The servers will be capable of handling high congestion.
- **Future developers:**
  Any developer will be able to work on the application due to the clean code rules that the application will follow and commented code and doc strings to explain what is happening.
- **Scalability:**
  The application will have a lot of room for scalability to keep up with any growth needs.

2.1.5.Usability Requirements

- **User friendly:**
  The application will have easy no cluttered web pages to ensure easy use and intuitive scrolling and shopping.
- **Up-to-date products:**
  Products will be of the highest quality and will be market leading.
- **Competitive pricing:**
  The products will not only be of the highest quality but they will be very competitively priced. With plenty of deals.
- **Easy user input:**
  Users will easily be able to input data as the application will be very intuitive. Forms will be clear and concise and will have adequate placeholder text to ensure the user knows what to enter. There will be alert messages to ensure the user knows what is happening at all times.
- **Design must be consistent:**

Design must be consistent throughout the whole application. Same colour schemes, fonts and image sizes on all web pages.

- **Feedback:**
  Users will be able to leave feedback on products via reviews. The user will be able to rate their experience in an email that will be sent out after each purchase.
- **Error handling:**
  The application will have error handling through to prevent errors been made during set or checkout. Alert messages will be present wherever important user input is needed.

2.2. Design & Architecture

Systems Architecture:

Star nutrition was built using Django a web framework that follows a Model-View-Tempelate architectural pattern. MVT is a variation of the Model-View-Controller pattern that is frequently used in the creation of online applications, especially in frameworks like Django. The three primary parts of the MVT pattern divide the application's concerns:

Model:

The Model shows the data and business logic of the application. It specifies how data is saved, retrieved, and altered, as well as its structure and behaviour. Models are often represented in Django applications as Python classes that correspond to database tables.

View:

The application's presentation logic is handled by the view. It is in charge of translating data from the Model into a readable format for users to view in their web browsers, typically HTML. Views in Django are Python classes or functions that accept HTTP requests and send HTTP answers.

Template:

The user interface's layout and structure must be specified by the Template. With placeholders for dynamic content, the HTML markup is contained in a distinct display layer. The display can be kept distinct from the underlying code and data processing thanks to templates. Django's template engine is used by templates to render dynamic material inside HTML.

Frontend(User Inerfaces):

The frontend is the graphical and interactive parts of the application that the end user will directly engage with when using Star nutrition. The front end is made up of a homepage, a base page, a products page, a product details page, a profile page, and a shopping cart page. The idea of these pages is so customers can explore products, examine product information, and prices, add products to their shopping cart, proceed to checkout, check previous orders, and personal details and tasks generally connected to an E-Commerce platform. For the front end, the use HTML, CSS and JavaScript were used to develop the application. Below are some details of the frontend of Star Nutrition.

Base Page:

The base.html file serves as a template for the entire web application's base layout. All other pages in the program are built upon the base.html file, which has common components including the header and navigation.

templates > base.html > html > head

```
<!--Allows static files to load-->
{% load static %}
<!doctype html>
<html lang="en">

<head>
    <!-- Required meta tags -->
    {% block meta %}
    <!--Allows support of older internet exploers-->
    <meta http-equiv="X-UA-Compatible" content="ie=edge">
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no">
    {% endblock %}


    {% block extra_meta %}
    {% endblock %}

    {% block corecss %}
    <link rel="stylesheet" href="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/css/bootstrap.min.css"
        integrity="sha384-Vkoo8x4CGsO3+Hhxv8T/Q5PaXtkKtu6ug5TOeNV6gBiFeWPGFN9MuhOf23Q9Ifjh" crossorigin="anonymous">
    <!-- add google fonts -->
    <link rel="stylesheet" href="https://fonts.googleapis.com/css?family=Lato&display=swap">
    <!-- linking static folder so css file can load -->
    <link rel="stylesheet" href="{% static 'css/base.css' %}">
    <!-- Stripe javascript to accept payments -->
    <script src="https://js.stripe.com/v3/"></script>
    {% endblock %}

    {% block extra_css %}
    {% endblock %}
```

templates > base.html > html > head

```
    {% block corejs %}
    <script src="https://code.jquery.com/jquery-3.4.1.slim.min.js"
        integrity="sha384-J6qa4849blE2+poT4WnyKhv5vZF5SrPo0iEjwBvKU7imGFAV0wwj1yYfoRSJoZ+n"
        crossorigin="anonymous"></script>
    <script src="https://code.jquery.com/jquery-3.4.1.min.js"
        integrity="sha256-CSXorXvZcTkaix6Yvo6HppcZGetbYMGWSFlBw8HfCJo=" crossorigin="anonymous"></script>
    <script src="https://cdn.jsdelivr.net/npm/popper.js@1.16.0/dist/umd/popper.min.js"
        integrity="sha384-Q6E9RHvbIyZFJoft+2mJbHaEWldlvI9IOYy5n3zV9zzTtmI3UksdQRVvoxMfooAo"
        crossorigin="anonymous"></script>
    <script src="https://stackpath.bootstrapcdn.com/bootstrap/4.4.1/js/bootstrap.min.js"
        integrity="sha384-wfSDF2E50Y2D1uUdj0O3uMBJnjuUD4Ih7YwaYd1iqfktj0Uod8GCExl3Og8ifwB6"
        crossorigin="anonymous"></script>
    <!-- add script for fontawseome  -->
    <script src="https://kit.fontawesome.com/e580420d0c.js" crossorigin="anonymous"></script>
    {% endblock %}

    {% block extra_js %}
    {% endblock %}

    <title>Star Nutrition {% block extra_title %}{% endblock %}</title>
</head>

<body>
    <!--header for the top of the bag nad make sure it sticks to the top of the page-->
    <header class="container-fluid fixed-top">
        <div id="topnav" class="row bg-white pt-lg-2 d-none d-lg-flex">
            <!--bootstrap so they stack vertacliyy on all screens and centred-->
            <div class="col-10 col-lg-4 my-auto py-1 py-lg-0 text-center text-lg-left">
                <!--links to the homepage url from urls.py-->
                <a href="{% url 'homepage' %}" class="nav-link main-logo-link">
                    <h2 class="logo-font text-black my-0"><strong>Star Nutrition</strong></h2>
                </a>
            </div>
            <div class="col-12 col-lg-4 my-auto py-1 py-lg-0">
                <!--Get method for searchs using url paramiters-->
                <form method="GET" action="{% url 'products' %}">
                    <!--input group for querying using q to be submitted to the action url-->
                    <div class="input-group w-100">
```

templates > base.html > html > body > header.container-fluid.fixed-top

```
        <!--input group for querying using q to be submitted to the action url-->
        <div class="input-group w-100">
            <input class="form-control border border-black rounded-0" type="text" name="q"
                placeholder="Search our site">
            <!--input group button using a search icon from front awesome-->
            <div class="input-group-append">
                <button class="form-control btn btn-black border border-black rounded-0" type="submit">
                    <span class="icon">
                        <i class="fas fa-search"></i>
                    </span>
                </button>
            </div>
        </div>
    </form>
</div>
<!--column for shopping bag and accoutns links-->
<div class="col-12 col-lg-4 my-auto py-1 py-lg-0">
    <!--bootstrap list inlines are from bootstarp documention-->
    <ul class="list-inline list-unstyled text-center text-lg-right my-0">
        <li class="list-inline-item dropdown">
            <!--Anchor acts as a parent menu for the font awesome icon and accoputn text-->
            <a class="text-black nav-link" href="#" id="user-options" data-toggle="dropdown"
                aria-haspopup="true" aria-expanded="false">
                <div class="text-center">
                    <!--dropdown menu that opens when clicked-->
                    <div><i class="fas fa-user fa-lg"></i></div>
                    <p class="my-0">My Account</p>
                </div>
            </a>
            <!--dropwdown meun using django tags for what to display depending on user access-->
            <div class="dropdown-menu border-0" aria-labelledby="user-options">
                {% if request.user.is_authenticated %}
                    {% if request.user.is_superuser %}
                        <a href="" class="dropdown-item">Product Management</a>
                    {% endif %}
                    <a href="{% url 'profile' %}" class="dropdown-item">My Profile</a>
                    <!--Alluath urls being used below-->
                    <a href="{% url 'account_logout' %}" class="dropdown-item">Logout</a>
                {% else %}
```

templates > base.html > html > body > header.container-fluid.fixed-top

```
                        <a href="{% url 'account_logout' %}" class="dropdown-item">Logout</a>
                    {% else %}
                        <a href="{% url 'account_signup' %}" class="dropdown-item">Register</a>
                        <a href="{% url 'account_login' %}" class="dropdown-item">Login</a>
                    {% endif %}
                </div>

            <li class="list-inline-item">
                <!--shopping bag link-->
                <a class="{% if grand_total %}text-info font-weight-bold{% else %}text-white{% endif %} nav-link"
                    href="{% url 'view_bag' %}">
                    <div class="text-center">
                        <div><i class="fas fa-shopping-bag fa-lg"></i></div>
                        <p class="my-0">
                            <!--Django template logic been used to check if grand total exists and execute and display toatl to 2 decimal paces-->
                            {% if grand_total %}
                            €{{ grand_total|floatformat:2 }}
                            <!--if the grand total template doesnt exist 0 will be displayed-->
                            {% else %}
                            €0.00
                            {% endif %}
                        </p>
                    </div>
                </a>
            </li>
        </ul>
    </div>
</div>
<div class="row bg-white">
    <!--toggle button for the main navigation-->
    <nav class="navbar navbar-expand-lg navbar-light w-100">
        <button class="navbar-toggler" type="button" data-toggle="collapse" data-target="#main-nav"
            aria-controls="main-nav" aria-expanded="false" aria-label="Toggle navigation">
            <span class="navbar-toggler-icon"></span>
        </button>
        {% include 'includes/mobile-header.html' %}
        {% include 'includes/main-nav.html' %}
    </nav>
</div>
```

```
templates > <> base.html > html > body > header.container-fluid.fixed-top
144             </div>
145 ⌄          <div id="delivery-banner" class="row text-center">
146 ⌄              <div class="col bg-black text-white">
147                    <h4 class="logo-font my-1">Free delivery on orders over €{{ free_delivery_threshold }}!</h4>
148                </div>
149            </div>
150        </header>
151
152        {% if messages %}
153 ⌄          <div class="message-container">
154                <!-- toast messages -->
155                <!-- taken from django docs, using with statment can check the level and return the apporirate toast -->
156                {% for message in messages %}
157                {% with message.level as level %}
158                    <!-- level 40 in django is an error -->
159                    {% if level == 40 %}
160                        {% include 'includes/toasts/toasts_error.html' %}
161                    <!-- level 30 is a warning -->
162                    {% elif level == 30 %}
163                        {% include 'includes/toasts/toasts_warning.html' %}
164                    <!-- level 25 is success -->
165                    {% elif level == 25 %}
166                        {% include 'includes/toasts/toast_success.html' %}
167                    <!-- defualt for everything else -->
168                    {% else %}
169                        {% include 'includes/toasts/toasts_info.html' %}
170                    {% endif %}
171                {% endwith %}
172                {% endfor %}
173            </div>
174        {% endif %}
175
176        {% block page_header %}
177        {% endblock %}
178
179        {% block content %}
180        {% endblock %}
181
182        {% block postloadjs %}
183        <!-- script for toast messages -->
184 ⌄      <script type="text/javascript">
185            $('.toast').toast('show');
186        </script>
187        {% endblock %}
188
```

{% load static %}:

The CSS, JavaScript, and image static files are loaded by this template tag. It enables the usage of the %static% template tag to refer to static assets later on in the file.

Block tags:

Several block tags in the template specify sections that can be changed in child templates. The material contained in these blocks can be replaced or expanded by child templates that extend the base.html template.

Meta Tags:

The template has meta tags for charset, viewport, and X-UA-Compatible. The behaviour and look of the web page are defined by these tags.

JavaScript and CSS:

Links to JavaScript and CSS stylesheets are present in the template. In this instance, it imports the base.css custom CSS file as well as the Bootstrap CSS and JavaScript library.

Font Awesome:

The template contains a link to the Font Awesome library, which offers a selection of icons for different elements.

Header:

The shopping bag link, search bar, user account dropdown, and site logo are all located in the header area. For layout and styling, Bootstrap classes are utilized.

Navigation:

The navigation area has two templates: includes/mobile-header.html and includes/main-nav.html, as well as a toggle button for mobile navigation (navbar-toggler). The navigation menus for desktop and mobile devices are probably defined individually in these templates.

Delivery Message:

The template has a banner that advertises free delivery for orders that exceed a specific amount.

Messages:

Toast messages, which are used to show user notifications or feedback messages (such as success, error, or warning messages), are displayed in this area. The messages template context variable allows access to the messages, which are normally set by the Django framework.

Main Nav:

The nav bar of Star Nutrition was designed using HTML. The code in the screenshot below can be found in the main templates folder, within the includes folder. It shows a menu bar with several drop-down menus, each with connections to various product categories.

```
main-nav.html
templates > includes > main-nav.html > div#main-nav.collapse.navbar-collapse > ul.navbar-nav.w-auto.mx-auto > li.nav-item.dropdown > div.dropdown-menu.border-0
1   <div class="collapse navbar-collapse" id="main-nav">
2       <ul class="navbar-nav w-auto mx-auto">
3           <li class="nav-item dropdown">
4               <!--products dropdown for links for user to access all products-->
5               <a class="logo-font font-weight-bold nav-link text-black mr-5" href="#" id="all-products-link" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
6                   All Products
7               </a>
8               <div class="dropdown-menu border-0" aria-labelledby="all-products-link">
9                   <!-- sort products by price in accending order -->
10                  <a href="{% url 'products' %}?sort=price&direction=asc" class="dropdown-item">By Price</a>
11                  <!-- sort products by rating in desending order -->
12                  <a href="{% url 'products' %}?sort=rating&direction=desc" class="dropdown-item ">By Rating</a>
13                  <!-- sort products by vategory in accending order -->
14                  <a href="{% url 'products' %}?sort=category&direction=asc" class="dropdown-item ">By Category</a>
15                  <!-- sort productsby all products -->
16                  <a href="{% url 'products' %}" class="dropdown-item">All Products</a>
17              </div>
18          </li>
19          <!--Vitamins & Supplements dropdown for links for user to access all Vitamins & Supplements-->
20          <li class="nav-item dropdown">
21              <a class="logo-font font-weight-bold nav-link text-black mr-5" href="#" id="supplement-link" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
22                  Vitamins & Supplements
23              </a>
24              <div class="dropdown-menu border-0" aria-labelledby="supplement-link">
25                  <a href="{% url 'products' %}?category=vitamins" class="dropdown-item">Vitamins</a>
26                  <a href="{% url 'products' %}?category=minerals" class="dropdown-item">Minerals</a>
27                  <a href="{% url 'products' %}?category=supplements" class="dropdown-item">Supplements</a>
28                  <a href="{% url 'products' %}?category=vitamins,minerals,supplements" class="dropdown-item">All Vitamins & Supplements</a>
29              </div>
30          </li>
```

```
        <!--Sports Nutrition dropdown for links for user to access Sports Nutrition-->
        <li class="nav-item dropdown">
            <a class="logo-font font-weight-bold nav-link text-black mr-5" href="#" id="online-link" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                Sports Nutrition
            </a>
            <div class="dropdown-menu border-0" aria-labelledby="online-link">
                <a href="{% url 'products' %}?category=protein_powders" class="dropdown-item">Protien Powders</a>
                <a href="{% url 'products' %}?category=bars_drinks_snacks" class="dropdown-item">Bars, Drinks, Snacks</a>
                <a href="{% url 'products' %}?category=creatine" class="dropdown-item">Creatine</a>
                <a href="{% url 'products' %}?category=electrolytes" class="dropdown-item">Electrolytes</a>
                <a href="{% url 'products' %}?category=energy_gels" class="dropdown-item">Energy Gels</a>
                <a href="{% url 'products' %}?category=amino_acids" class="dropdown-item">Amino Acids</a>
                <a href="{% url 'products' %}?category=sports_accessories" class="dropdown-item">Sports Accessories</a>
                <a href="{% url 'products' %}category=protein_powders,bars_drinks_snacks,creatine,electrolytes,energy_gels,amino_acids,sports_accessories" class="dropdown-item">A
            </div>
        </li>
        <!--Weight management dropdown for links for user to access all Weight management products-->
        <li class="nav-item dropdown">
            <a class="logo-font font-weight-bold nav-link text-black mr-5" href="#" id="clearance-link" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
                Weight Management
            </a>
            <div class="dropdown-menu border-0" aria-labelledby="clearance-link">
                <a href="{% url 'products' %}?category=fat_burner" class="dropdown-item">Fat Burners</a>
                <a href="{% url 'products' %}?category=weight_management" class="dropdown-item">Weight Management</a>
                <a href="{% url 'products' %}?category=foods_drinks" class="dropdown-item">Diet Drinks and Foods</a>
                <a href="{% url 'products' %}?category=fat_burner,weight_management,foods_drinks" class="dropdown-item">All Weight Management products</a>
            </div>
        </li>
    </ul>
</div>
```

All Products Dropdown:

Dropdown Content:

- "By Price" link: Sorts products by price in ascending order when clicked.
- "By Rating" link: Sorts products by rating in descending order when clicked.
- "By Category" link: Sorts products by category in ascending order when clicked.
- "All Products" link: Displays all products when clicked.

Vitamins & Supplements Dropdown:

Dropdown Content:

- "Vitamins" link: Displays products in the "Vitamins" category when clicked.
- "Minerals" link: Displays products in the "Minerals" category when clicked.
- "Supplements" link: Displays products in the "Supplements" category when clicked.
- "All Vitamins & Supplements" link: Displays all products in the "Vitamins," "Minerals," and "Supplements" categories when clicked.

Sports Nutrition Dropdown:

Dropdown Content:

- "Protein Powders" link: Displays products in the "Protein Powders" category when clicked.
- "Bars, Drinks, Snacks" link: Displays products in the "Bars, Drinks, Snacks" category when clicked.
- "Creatine" link: Displays products in the "Creatine" category when clicked.
- "Electrolytes" link: Displays products in the "Electrolytes" category when clicked.
- "Energy Gels" link: Displays products in the "Energy Gels" category when clicked.
- "Amino Acids" link: Displays products in the "Amino Acids" category when clicked.
- "Sports Accessories" link: Displays products in the "Sports Accessories" category when clicked.
- "All Sports Nutrition Products" link: Displays all products in the sports nutrition categories when clicked.

Weight Management Dropdown:

Dropdown Content:

- "Fat Burners" link: Displays products in the "Fat Burners" category when clicked.
- "Weight Management" link: Displays products in the "Weight Management" category when clicked.
- "Diet Drinks and Foods" link: Displays products in the "Diet Drinks and Foods" category when clicked.
- "All Weight Management products" link: Displays all products in the weight management categories when clicked.

Some of the code contains Django template tags that URLs in Start Nutrion to link to certain products for example "products" URL pattern is set eg. "% url 'products'%" this will bring the user to all the products pages.

```
<div class="dropdown-menu border-0" aria-labelledby="supplement-link">
    <a href="{% url 'products' %}?category=vitamins" class="dropdown-item">Vitamins</a>
    <a href="{% url 'products' %}?category=minerals" class="dropdown-item">Minerals</a>
    <a href="{% url 'products' %}?category=supplements" class="dropdown-item">Supplements</a>
    <a href="{% url 'products' %}?category=vitamins,minerals,supplements" class="dropdown-item">All Vitamins & Supplements</a>
</div>
```

Anchor tags <a>, which indicate the target URLs for each category or sorting option, are used to build the links in each dropdown. The navigation bar and dropdown menus are also styled using Bootstrap classes.

Wireframe:

Logo
Seach bar
Profile
Bag

Screenshot:

STAR NUTRITION
Search our site
My Account
€74.80
ALL PRODUCTS
VITAMINS & SUPPLEMENTS
SPORTS NUTRITION
WEIGHT MANAGEMENT

Mobile Header:

The mobile header is designed using HTML. The code represents a navigation bar for mobile devices that includes a search bar, user account dropdown, and shopping bag display, among other things.

```
mobile-header.html
templates > includes > mobile-header.html > li.list-inline-item > a#mobile-search.text-black.nav-link.d-block.d-lg-none
1  <!--Button to open the search bar-->
2  <li class="list-inline-item">
3      <a class="text-black nav-link d-block d-lg-none" href="#" id="mobile-search" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
4          <div class="text-center">
5              <div><i class="fas fa-search fa-lg"></i></div>
6              <p class="my-0">Search</p>
7          </div>
8      </a>
9      <!--Search form now in displayed in a dropdown for moblie devices-->
10     <div class="dropdown-menu border-0 w-100 p-3 rounded-0 my-0" aria-labelledby="mobile-search">
11         <!--Get method for searchs using url paramiters-->
12         <form class="form" method="GET" action="{% url 'products' %}">
13             <div class="input-group w-100">
14                 <!--From used instead of a list of links-->
15                 <input class="form-control border border-black rounded-0" type="text" name="q" placeholder="Search our site">
16                 <div class="input-group-append">
17                     <button class="form-control form-control btn btn-black border border-black rounded-0" type="submit">
18                         <span class="icon">
19                             <i class="fas fa-search"></i>
20                         </span>
21                     </button>
22                 </div>
23             </div>
24         </form>
25     </div>
26 </li>
27 <!--Button to open the my account Login dropdown menu-->
28 <li class="list-inline-item dropdown">
29     <a class="text-black nav-link d-block d-lg-none" href="#" id="user-options" data-toggle="dropdown" aria-haspopup="true" aria-expanded="false">
30         <div class="text-center">
31             <div><i class="fas fa-user fa-lg"></i></div>
32             <p class="my-0">My Account</p>
33         </div>
34     </a>
35
36 </li>
```

```
<!--Dropwdon to acces the shopping bag on mobile-->
<li class="list-inline-item">
    <a class="{% if grand_total %}text-primary font-weight-bold{% else %}text-black{% endif %} nav-link d-block d-lg-none" href="{% url 'view_bag' %}">
        <div class="text-center">
            <div><i class="fas fa-shopping-bag fa-lg"></i></div>
            <p class="my-0">
                {% if grand_total %}
                    ${{ grand_total|floatformat:2 }}
                {% else %}
                    $0.00
                {% endif %}
            </p>
        </div>
    </a>
</li>
```

Search Bar Dropdown:

Button:

On mobile devices, a search icon with the word "Search" is visible.

Display Content:

- Search Form: A dropdown menu containing a search form appears when the search icon is clicked.
- Form Method: The product URL is reached using the GET method, which sends the search query as a URL and will bring the user to the product page.
- Text Input: The search query is entered into the "q"-named text input area.
- A search icon button is used as the form's submit button.

User Account Dropdown:

- On mobile devices, a user icon with the label "My Account" is visible.

Shopping Bag Display:

- On mobile devices, a shopping bag icon is displayed along with the total cost of the things inside the bag.
- Conditional Styling: Depending on whether or not the shopping bag contains items, the icon and total price are styled differently. The grand_total is a variable expressing the total price of the items in your bag.

Homepage:

Star Nutrition’s home page serves as its landing page. To aid consumers in finding what they're looking for fast, it consists of all the platform's products, shopping bags, delivery information, and a search bar.

This file is located in the index.html file in the homepage app. This page acts as a child to the base.html file as it extends it. It uses Django template logic to load certain blocks from the base.html file. Below is a screen shot of the index.html file.

```
homepage > templates > home > <> index.html > ...
1   {% extends "base.html" %}
2   {% load static %}
3
4   {% block page_header %}
5       <div class="container header-container">
6           <div class="row">
7               <div class="col"></div>
8           </div>
9       </div>
10  {% endblock %}
11
12  {% block content %}
13      <!--home page container HTML using bootstrap-->
14      <div class="container h-100">
15          <div class="row h-100">
16              <div class="col-7 col-md-6 my-auto">
17                  <h1 class="display-4 logo-font text-black">
18                      Start your journey to a better quility of life now!
19                  </h1>
20                  <div class="my-4">
21                      <h4>
22                          <a href="{% url 'products' %}" class="shop-now-button btn btn-lg rounded-0 text-uppercase py-2">Lets get started!</a>
23                      </h4>
24                  </div>
25              </div>
26          </div>
27      </div>
28  {% endblock %}
```

{% extends "base.html" %}:

This line instructs Django that the base.html template should be used as a basis for this template. It indicates that while this template can override particular blocks as necessary, it will use the layout and structure established in base.html.

{% load static %}:

Similar to the base.html template, this template element loads static files.

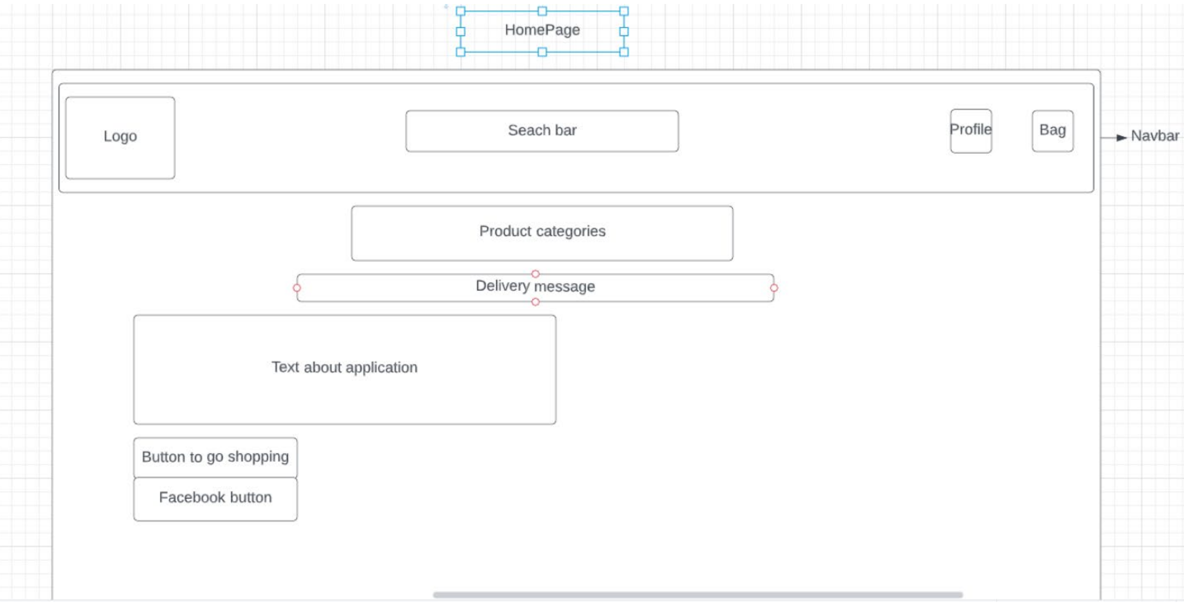
{% block page_header %}:

This child template replaces a block that is defined in the base.html template. The content from this child template is used to replace the page_header block's content in the base template. In this instance, it expands the header section to include a container with a row and column.

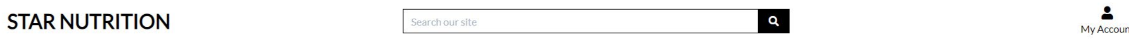
{% block content %}:

This child template replaces a block that is also defined in the base.html template. The content from this child template is used to replace the material in the base template's content block. In this instance, it uses Bootstrap classes to create a container with a row and column. There is a heading and a "Shop Now" button in the column.

Wireframe:



Actual screenshot



Products Page:

The products.html file extends the base.html file. The products page renders as multiple blocks. The page was designed using HTML, and bootstrap CSS using Django template logic that renders the product details.

```
products.html ×
products > templates > products > products.html > div.container-fluid > div.row > div.product-container.col-10.offset-1 > div.row.mt-1.mb-2 > div.col-12.col-md-6.my-auto.order-md-last.d-flex.justify-content-center.justify-content-md
1   {% extends "base.html" %}
2   {% load static %}
3
4   {% block page_header %}
5       <div class="container header-container">
6           <div class="row">
7               <div class="col"></div>
8           </div>
9       </div>
10  {% endblock %}
11
```

{% extends "base.html" %}:

This line instructs Django that the base.html template should be used as a basis for this template. It indicates that while this template can override particular blocks as necessary, it will use the layout and structure established in base.html.

{% load static %}:

Similar to the base.html template, this template element loads static files.

{% block page_header %}:

This child template replaces a block that is defined in the base.html template. The content from this child template is used to replace the page_header block's content in the base template. In this instance, it expands the header section to include a container with a row and column.

```
{% block content %}
<!-- class to cover the projects background imgae and give it a great background -->
<div class="overlay"></div>
    <div class="container-fluid">
        <!-- row for products -->
        <div class="row">
            <div class="col text-center mt-3">
                <h2 class="logo-font text-black">Products</h2>
                <!-- bootstrap template to render category links -->
                {% for c in current_categories %}
                    <a class="category-badge text-decoration-none" href="{% url 'products' %}?category={{ c.name }}">
                        <span class="p-2 mt-2 badge badge-white text-black rounded-0 border border-dark">{{ c.friendly_name }}</span>
                    </a>
                {% endfor %}
                <hr class="w-50 mb-1">
            </div>
        </div>
        <div class="row">
            <div class="product-container col-10 offset-1">
                <div class="row mt-1 mb-2">
                    <!-- sort layout box on top  -->
                    <div class="col-12 col-md-6 my-auto order-md-last d-flex justify-content-center justify-content-md-end">
                        <div class="sort-select-wrapper w-50">
                            <select id="sort-selector" class="custom-select custom-select-sm rounded-0 border border-{% if current_sorting != 'None_None' %}info{% else %}black{%
                                <!-- sorting methods depening on selected metho -->
                                <option value="reset" {% if current_sorting == 'None_None' %}selected{% endif %}>Sort by...</option>
                                <option value="price_asc" {% if current_sorting == 'price_asc' %}selected{% endif %}>Price (low to high)</option>
                                <option value="price_desc" {% if current_sorting == 'price_desc' %}selected{% endif %}>Price (high to low)</option>
                                <option value="rating_asc" {% if current_sorting == 'rating_asc' %}selected{% endif %}>Rating (low to high)</option>
                                <option value="rating_desc" {% if current_sorting == 'rating_desc' %}selected{% endif %}>Rating (high to low)</option>
                                <option value="name_asc" {% if current_sorting == 'name_asc' %}selected{% endif %}>Name (A-Z)</option>
                                <option value="name_desc" {% if current_sorting == 'name_desc' %}selected{% endif %}>Name (Z-A)</option>
                                <option value="category_asc" {% if current_sorting == 'category_asc' %}selected{% endif %}>Category (A-Z)</option>
                                <option value="category_desc" {% if current_sorting == 'category_desc' %}selected{% endif %}>Category (Z-A)</option>
                            </select>
                        </div>
                    </div>
                    <!-- sort results to show count of results -->
                    <div class="col-12 col-md-6 order-md-first">
                        <p class="text-muted mt-3 text-center text-md-left">
                            <!-- django template logic -->
                            {% if search_term or current_categories or current_sorting != 'None_None' %}
                                <span class="small"><a href="{% url 'products' %}">Products Home</a> | </span>
```

```
54                                  <span class="small"><a href="{% url 'products' %}">Products Home</a> | </span>
55                              {% endif %}
56                              {{ products|length }} Products{% if search_term %} found for <strong>"{{ search_term }}"</strong>{% endif %}
57                          </p>
58                      </div>
59                  </div>
60                      <!-- dajngo for loop used to iterate through the products valiable -->
61                      <div class="row">
62                          {% for product in products %}
63                          <!-- for each product a column will be generate -->
64                              <div class="col-sm-6 col-md-6 col-lg-4 col-xl-3">
65                                  <!-- bootstraps cards used to diaply product -->
66                                  <div class="card h-100 border-0">
67                                      <!-- top of the bootsrap card which contains the image of the product -->
68                                      <!-- if statment using images url that calls the image url -->
69                                      {% if product.image %}
70                                      <!-- link to project details page -->
71                                      <a href="{% url 'product_detail' product.id %}">
72                                          <img class="card-img-top img-fluid" src="{{ product.image.url }}" alt="{{ product.name }}">
73                                      </a>
74                                      {% else %}
75                                      <!-- link to project details page -->
76                                      <a href="{% url 'product_detail' product.id %}">
77                                          <img class="card-img-top img-fluid" src="{{ MEDIA_URL }}noimage.png" alt="{{ product.name }}">
78                                      </a>
79                                      {% endif %}
80                                      <!-- card boy which contains the products name-->
81                                      <div class="card-body pb-0">
82                                          <p class="mb-0">{{ product.name }}</p>
83                                      </div>
84                                      <!-- footer which will containt the products details -->
85                                      <div class="card-footer bg-white pt-0 border-0 text-left">
86                                          <div class="row">
87                                              <div class="col">
88                                                  <!-- calls product pricing -->
89                                                  <p class="lead mb-0 text-left font-weight-bold">€{{ product.price }}</p>
90                                                  <!-- product catgory section for sorting  -->
91                                                  {% if product.category %}
92                                                  <p class="small mt-1 mb-0">
93                                                      <a class="text-muted" href="{% url 'products' %}?category={{ product.category.name }}">
94                                                          <!-- render it by catgory and friendly name -->
95                                                          <i class="fas fa-tag mr-1"></i>{{ product.category.friendly_name }}
96                                                      </a>
```

```
96                              </a>
97                          </p>
98                          {% endif %}
99                          <!-- calls product rating -->
100                         {% if product.rating %}
101                             <small class="text-muted"><i class="fas fa-star mr-1"></i>{{ product.rating }} / 5</small>
102                         {% else %}
103                             <small class="text-muted">No Rating</small>
104                         {% endif %}
105                     </div>
106                 </div>
107             </div>
108         </div>
109     </div>
110     <!-- horizontal rule to divide the products on all page sizes -->
111     <!-- if statment for divisible by 1 then render this column -->
112     {% if forloop.counter|divisibleby:1 %}
113         <!-- small to medium screens -->
114         <div class="col-12 d-sm-none mb-5">
115             <hr>
116         </div>
117     {% endif %}
118     <!-- if statment for divisible by 2 then render this column -->
119     {% if forloop.counter|divisibleby:2 %}
120         <!-- hidden on extra small screens and dipslay block on small screens or larger -->
121         <div class="col-12 d-none d-sm-block d-md-block d-lg-none mb-5">
122             <hr>
123         </div>
124     {% endif %}
125     <!-- if statment for divisible by 3 then render this column -->
126     {% if forloop.counter|divisibleby:3 %}
127         <!-- hidden on everything but large screens -->
128         <div class="col-12 d-none d-lg-block d-xl-none mb-5">
129             <hr>
130         </div>
131     {% endif %}
132     <!-- if statment for divisible by 4 then render this column -->
133     {% if forloop.counter|divisibleby:4 %}
134         <!-- xtra large screens -->
135         <div class="col-12 d-none d-xl-block mb-5">
136             <hr>
137         </div>
138     {% endif %}
```

```
138                     {% endif %}
139                 {% endfor %}
140             </div>
141         </div>
142     </div>
143     </div>
144     <div class="btt-button shadow-sm rounded-0 border border-black">
145         <a class="btt-link d-flex h-100">
146             <i class="fas fa-arrow-up text-black mx-auto my-auto"></i>
147         </a>
148     </div>
149 {% endblock %}
150
```

{% block content %}:

Overlay and Container:

To produce a backdrop effect, a div with the class "overlay" is used as an overlay.

A "container-fluid" div is used to enclose the content, ensuring that it fills the viewport's entire width.

Categories Section:

The first piece of the code shows product categories as badges. The badges act as links to products that can be filtered by the chosen category.

Sorting Section:

The dropdown selects an element in the following section and is used to order the shown products based on the following price, rating, name, and category.

Product Listing:

- The majority of the template is dedicated to displaying the products in a grid layout.
- The image, name, price, category, and rating of each product are displayed on a Bootstrap card.
- The products' details are dynamically filled in when they are iterated using a Django for a loop.
- The template contains conditional statements that determine whether a product has a picture, and if not, displays "noimage.png" by default.

- Horizontal lines (hr tags) are used to create a visual separation between rows of products on different screen sizes depending on the position of the product in the for loop.

The back-to-top button:

The button at the bottom of the template enables users to swiftly scroll back to the top of the page.

```
    </script>

    <script type="text/javascript">
        // captruing change event from selector
        $('#sort-selector').change(function() {
            var selector = $(this);
            var currentUrl = new URL(window.location);

            var selectedVal = selector.val();
            // resets sorting to default
            if(selectedVal != "reset"){
                // if not sorted and directions gets split at underscore by sorting or direction
                var sort = selectedVal.split("_")[0];
                var direction = selectedVal.split("_")[1];

                // replace get parameters in the url
                currentUrl.searchParams.set("sort", sort);
                currentUrl.searchParams.set("direction", direction);

                // replace current location to the updated url and relaod page
                window.location.replace(currentUrl);
            } else {
                currentUrl.searchParams.delete("sort");
                currentUrl.searchParams.delete("direction");

                window.location.replace(currentUrl);
            }
        })
    </script>
{% endblock %}
```

{% block postloadjs %} and {% endblock %}:

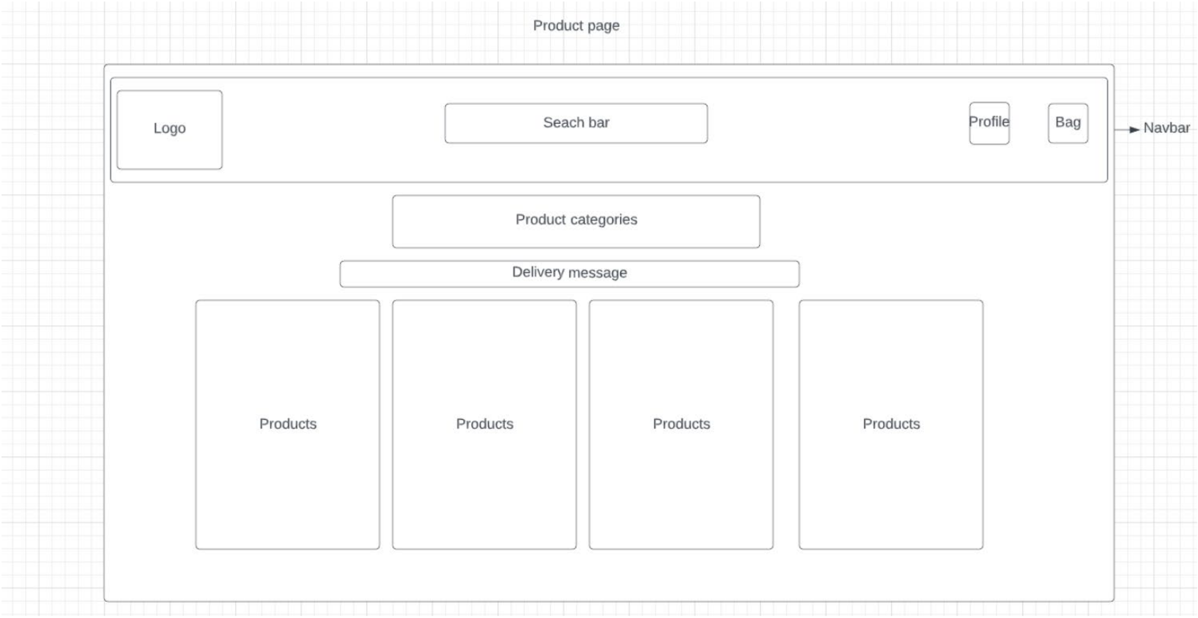
Back to Top Button:

- A smooth "Back to Top" effect is produced when the user clicks on the element with the class ".btt-link," which causes JavaScript code to run and scroll the window to the top.
- When a user has scrolled down to look at products, this functionality makes it simple for them to return to the top of the page.

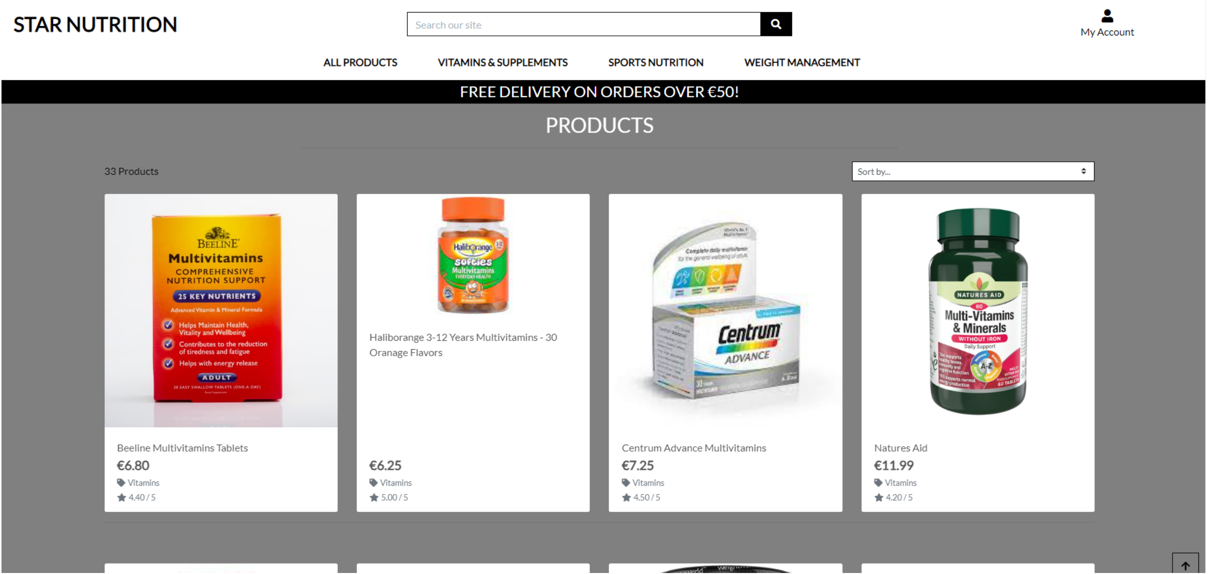
Sorting Functionality:

- The JavaScript code runs the sorting feature when the user chooses a different choice from the #sort-selector dropdown without requiring a page reload.
- It records the drop-down's change event ($('#sort-selector').change(...)) and returns the chosen value, such as "price_asc" for ascending price sorting.
- If the selected value is not "reset," it uses the URLSearchParams API to change the URL parameters for sorting and direction by extracting the sorting method and direction from the selected value.
- The sorting and direction parameters are removed from the URL if the specified value is "reset."
- To update the page content based on the chosen sorting option without reloading the full page, it finally replaces the existing URL with the updated URL.

Wireframe



Screenshot:



Products Details Page:

The product_detail.html file extends the base.html file. The products detail page renders the details of the selected product. It is a user-friendly user interface for viewing product information and allows users to add products to their shopping cart.

```
product_detail.html ×
products > templates > products > <> product_detail.html > ...
1   {% extends "base.html" %}
2   {% load static %}
3
4   {% block page_header %}
5   <div class="container header-container">
6       <div class="row">
7           <div class="col"></div>
8       </div>
9   </div>
10  {% endblock %}
```

{% extends "base.html" %}:

- This line instructs Django that the base.html template should be used as a basis for this template. It indicates that while this template can override particular blocks as necessary, it will use the layout and structure established in base.html.

{% load static %}:

- Similar to the base.html template, this template element loads static files.

{% block page_header %}:

- This child template replaces a block that is defined in the base.html template. The content from this child template is used to replace the page_header block's content in the base template. In this instance, it expands the header section to include a container with a row and column.

```
12  {% block content %}
13  <!-- class to cover the projects background imgae and give it a great background -->
14  <div class="overlay"></div>
15  <div class="container-fluid">
16      <!-- row for products -->
17      <!-- one row needed to be split into 2 columns to display product details -->
18      <div class="row">
19          <!-- col 1 displays product image -->
20          <div class="col-12 col-md-6 col-lg-4 offset-2">
21              <div class="image-container my-5">
22                  <!-- taken from products page and slightly updated -->
23                  <!-- top of the bootsrap card which contains the image of the product -->
24                  <!-- if statment using images url that calls the image url -->
25                  {% if product.image %}
26                  <a href="{{ product.image.url }}" target="_blank">
27                      <img class="card-img-top img-fluid" src="{{ product.image.url }}" alt="{{ product.name }}">
28                  </a>
29                  {% else %}
30                  <a href="">
31                      <img class="card-img-top img-fluid" src="{{ MEDIA_URL }}noimage.png" alt="{{ product.name }}">
32                  </a>
33                  {% endif %}
34              </div>
35          </div>
36          <!-- add products detials section  -->
37          <div class="col-12 col-md-6 col-lg-4">
38              <div class="product-details-container mb-5 mt-md-5">
39                  <!-- return product name -->
40                  <p class="mb-0">{{ product.name }}</p>
41                  <!-- return product price -->
42                  <p class="lead mb-0 text-left font-weight-bold">€{{ product.price }}</p>
43                  <!-- product catgory section for sorting  -->
44                  {% if product.category %}
45                  <p class="small mt-1 mb-0">
46                      <a class="text-muted" href="{% url 'products' %}?category={{ product.category.name }}">
47                          <!-- render it by catgory and friendly name -->
48                          <i class="fas fa-tag mr-1"></i>{{ product.category.friendly_name }}
49                      </a>
50                  </p>
51                  {% endif %}
52                  <!-- return product rating -->
53                  {% if product.rating %}
54                  <small class="text-muted"><i class="fas fa-star mr-1"></i>{{ product.rating }} / 5</small>
55                  {% else %}
```

```
products > templates > products > <> product_detail.html > ...
                    <small class="text-muted">No Rating</small>
                    {% endif %}
                    <!-- return product description -->
                    <p class="mt-3">{{ product.description }}</p>
                    <!-- from to use post method for adding products to the bag -->
                    <form class="form" action="{% url 'add_to_bag' product.id %}" method="POST">
                        <!-- token to stop cross sight security, cannot submit witout the token -->
                        {% csrf_token %}
                        <!-- form with one row split into two columns -->
                        <div class="form-row">
                            <!-- add sizes selector box to product detials template -->
                            {% with product.has_sizes as s %}
                            {% if s %}
                            <div class="col-12">
                                <!-- form for selecting size -->
                                <p><strong>Size:</strong></p>
                                <select class="form-control rounded-0 w-50" name="product_size" id='id_product_size'>
                                    <option value="xs">XS</option>
                                    <option value="s">S</option>
                                    <!-- Medium will be the defualt selection -->
                                    <option value="m" selected>M</option>
                                    <option value="l">L</option>
                                    <option value="xl">XL</option>
                                </select>
                            </div>
                            {% endif %}
                            <!-- column one takes the input for thee quanity of items they want to purchase -->
                            <div class="col-12">
                                <p class="mt-3"><strong>Quantity:</strong></p>
                                <div class="form-group w-50">
                                    <div class="input-group">
                                        <!-- add - button for taking away from item quantity -->
                                        <div class="input-group-prepend">
                                            <!-- buttons for javascript -->
                                            <button class="decrement-qty btn btn-black rounded-0"
                                            data-item_id="{{ product.id }}" id="decrement-qty_{{ product.id }}">
                                            <span class="icon">
                                                <i class="fas fa-minus"></i>
                                            </span>
                                            </button>
                                        </div>
                                        <input class="form-control qty_input" type="number" name="quantity" value="1"
                                            min="1" max="99" data-item_id="{{ product.id }}" id="id_qty_{{ product.id }}">
                                        <!-- add button for + button for adding to the product quanity -->
                                        <div class="input-group-append">
```

```
products > templates > products > <> product_detail.html > ...
                                            <!-- buttons for javascript -->
                                            <button class="increment-qty btn btn-black rounded-0"
                                                    data-item_id="{{ product.id }}" id="increment-qty_{{ product.id }}">
                                                    <span class="icon">
                                                        <i class="fas fa-plus"></i>
                                                    </span>
                                                </button>
                                        </div>
                                    </div>
                                </div>
                            </div>
                            <!-- column two handles the submit form for adding products to the bag -->
                            <!-- add full width and margin if product has margins -->
                            <div class="col{% if s %}-12 {% endif %}">
                                <!-- button to contoinue shopping and link back to the products page if clicked-->
                                <a href="{% url 'products' %}" class="btn btn-outline-black rounded-0 mt-5">
                                    <span class="icon">
                                        <i class="fas fa-chevron-left"></i>
                                    </span>
                                    <span class="text-uppercase">Continue shopping</span>
                                </a>
                                <!-- submit button to add products to the bag -->
                                <input type="submit" class="btn btn-black rounded-0 text-uppercase mt-5" value="Add to Bag">
                            </div>
                            <!-- hidden field for redirct url to go back to last page -->
                            <input type="hidden" name="redirect_url" value="{{ request.path }}">
                            {% endwith %}
                        </div>
                    </form>
                </div>
            </div>
        </div>
    </div>
```

{% block content %}:

Overlay and Container:

- To produce a backdrop effect, a div with the class "overlay" is used as an overlay.
- A "container-fluid" div is used to enclose the content, ensuring that it fills the viewport's entire width.

Product Details Layout:

- The product details are displayed using a Bootstrap row with two columns: one for the product image and one for the product information.
- The product image is displayed in the first column (col-12 col-md-6 col-lg-4 offset-2). The offset-2 class centres the column horizontally on smaller screens.
- The information about the product, including its name, price, category, rating, and description, is shown in the second column (col-12 col-md-6 col-lg-4).
- Product Image:
- The image of the product is presented inside a Bootstrap card-like container with the class "image-container."

- An if statement determines whether the item has a picture. If it does, the image is displayed with an anchor link that, when clicked, launches the image in a new tab. If not, the default picture (noimage.png) is shown.

Product Details:

- HTML components and Django template tags are used to display product information such as name, price, category, and rating.
- If a category exists for the product, it is shown as a link that, when clicked, filters the products according to the chosen category.

Form to Add to Bag:

- The product is added via a form to the shopping bag (cart).
- With the product's ID as a parameter, the form action is set to the URL pattern "add_to_bag".
- The form has a security feature called a CSRF token (%csrf_token%) that guards against cross-site request forgery attacks.
- Users can choose the quantity of the product they want to put in the bag using the form.
- A size selector dropdown is present if the product (product.has_sizes) has various sizes.
- Additionally, the form has buttons (increment-qty and decrement-qty) that allow you to change the quantity using JavaScript.

Continue Shopping Button:

- A "Continue Shopping" button with a link to the products page (% url 'products'%) is located beneath the quantity selector.

Hidden Field for Redirect URL:

- The redirect URL, which enables customers to go back to the current product information page after adding the item to the bag, is kept in a hidden field.

```
138 <script type="text/javascript">
139
140     // Disable +/- buttons outside 1-99 range
141     function handleEnableDisable(itemId) {
142         var currentValue = parseInt($(`#id_qty_${itemId}`).val());
143         var minusDisabled = currentValue < 2;
144         var plusDisabled = currentValue > 98;
145         $(`#decrement-qty_${itemId}`).prop('disabled', minusDisabled);
146         $(`#increment-qty_${itemId}`).prop('disabled', plusDisabled);
147     }
148
149     // Ensure proper enabling/disabling of all inputs on page load
150     var allQtyInputs = $('.qty_input');
151     for(var i = 0; i < allQtyInputs.length; i++){
152         var itemId = $(allQtyInputs[i]).data('item_id');
153         handleEnableDisable(itemId);
154     }
155
156     // Check enable/disable every time the input is changed
157     $('.qty_input').change(function() {
158         var itemId = $(this).data('item_id');
159         handleEnableDisable(itemId);
160     });
161
162     // Increment quantity
163     $('.increment-qty').click(function(e) {
164        e.preventDefault();
165        var closestInput = $(this).closest('.input-group').find('.qty_input')[0];
166        var currentValue = parseInt($(closestInput).val());
167        $(closestInput).val(currentValue + 1);
168        var itemId = $(this).data('item_id');
169        handleEnableDisable(itemId);
170     });
171
172     // Decrement quantity
173     $('.decrement-qty').click(function(e) {
174        e.preventDefault();
175        var closestInput = $(this).closest('.input-group').find('.qty_input')[0];
176        var currentValue = parseInt($(closestInput).val());
177        $(closestInput).val(currentValue - 1);
178        var itemId = $(this).data('item_id');
179        handleEnableDisable(itemId);
180     });
181 </script>
```

{% block postloadjs %}

handleEnableDisable Function:

- This function accepts an itemId as an argument, which stands for the distinct product identifier that should be used to handle the quantity input and buttons.
- It obtains the most recent quantity input value for the product.
- Based on the current value, it determines whether the - (decrement) and + (increment) buttons should be disabled. The - button is disabled if the value is less than 2, while the + button is disabled if the value is more than 98.
- Using jQuery's prop method, the buttons' disabled property is set appropriately.

Ensure Proper Enabling/Disabling on Page Load:

- The class provides the code with access to all quantity input elements. the allQtyInputs variable and selects them with the jQuery selector.
- It retrieves the relevant itemId after iterating through each input element.
- To make sure the buttons are enabled or disabled correctly when the page loads, it calls the handleEnableDisable function for each itemId.

Check Enable/Disable on Quantity Change:

- All amount input elements with the class have a change event listener added by the code.qty_input.
- The event handler is activated when the quantity input value changes.
- To update the button states based on the updated amount value, it fetches the itemId linked to the modified input and uses the handleEnableDisable method.

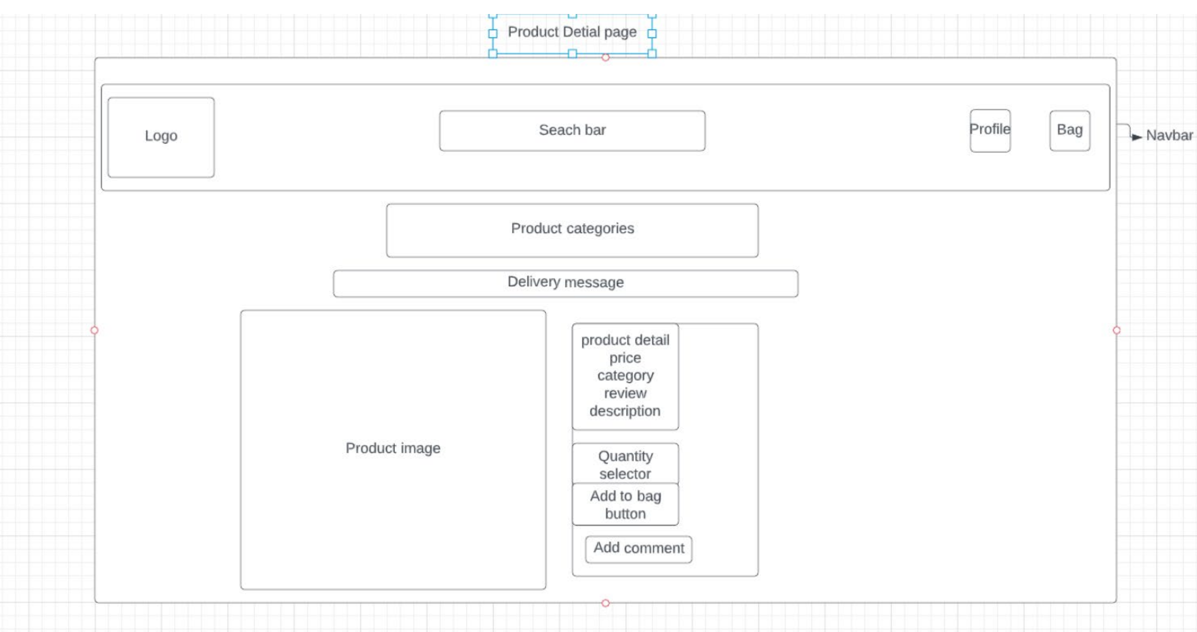
Increment Quantity:

- To all items with the class.increment-qty, which are the + buttons, the code adds a click event listener.
- The event handler is activated when a plus button is clicked.
- Using jQuery's closest method, it locates the nearest amount of input element and returns the related itemId.
- It invokes the handleEnableDisable function to update the button states based on the new amount value after increasing the input element's value by 1 and calling handleEnableDisable.

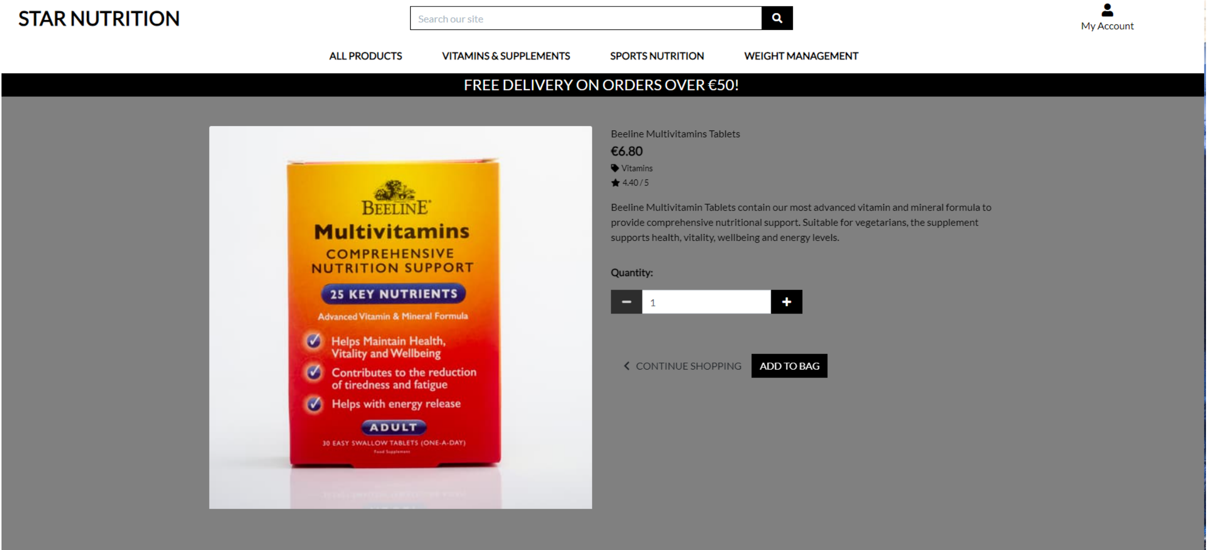
Decrement Quantity:

- To all items with the class.decrement-qty, which are the - buttons, the code adds a click event listener.
- The event handler is activated when a - button is clicked.
- Using jQuery's closest method, it locates the nearest amount of input element and returns the related itemId.
- It invokes the handleEnableDisable function to change the button states depending on the new amount value and decreases the input element's value by 1 before doing so.

Wireframe



Screenshot:



Profile:

The profile.html file extends the base.html file. The profiles page renders the user's profile page with delivery and order history. The page was designed using HTML, and bootstrap CSS using Django template logic that renders the profile details.

```
{% extends "base.html" %}
{% load static %}
<!-- adding css box for the template of the app -->
{% block extra_css %}
<link rel="stylesheet" href="{% static 'profiles/css/profiles.css' %}">
{% endblock %}

{% block page_header %}
<div class="container header-container">
    <div class="row">
        <div class="col"></div>
    </div>
</div>
{% endblock %}
```

{% extends "base.html" %}:

- This line instructs Django that the base.html template should be used as a basis for this template. It indicates that while this template can override particular blocks as necessary, it will use the layout and structure established in base.html.

{% load static %}:

- Similar to the base.html template, this template element loads static files.

{% block page_header %}:

- This child template replaces a block that is defined in the base.html template. The content from this child template is used to replace the page_header block's content in the base template. In this instance, it expands the header section to include a container with a row and column.

```
profile.html ×
profiles > templates > profiles > profile.html > ...
14  {% endblock %}
15
16  {% block content %}
17  <!-- covers background image -->
18  <div class="overlay"></div>
19  <div class="container">
20      <!-- row to contain shopping bag heading -->
21      <div class="row">
22          <div class="col">
23              <hr>
24              <h2 class="logo-font mb-4">Profile</h2>
25              <hr>
26          </div>
27      </div>
28      <div class="col-12 col-lg-6">
29          <!-- user delivery information in column 1 -->
30          <p class="text-muted">Default Delivery Information</p>
31          <!-- link to the profile url using a post method -->
32          <form class="mt-3" action="{% url 'profile' %}" method="POST" id="profile-update-form">
33              {% csrf_token %}
34              {{ form|crispy }}
35              <!-- submit button floated to the right of the column -->
36              <button class="btn btn-black rounded-0 text-uppercase float-right">Update Information</button>
37          </form>
38      </div>
39      <!-- user order history -->
40      <div class="col-12 col-lg-6">
41          <p class="text-muted">Order History</p>
42          <div class="order-history table-responsive">
43              <!-- making the table small and have no baorder -->
44              <table class="table table-sm table-borderless">
45                  <thead>
46                      <tr>
47                          <!-- four items in the table -->
48                          <th>Order Number</th>
49                          <th>Date</th>
50                          <th>Items</th>
51                          <th>Order Total</th>
52                      </tr>
53                  </thead>
54                  <tbody>
55                      <!-- for loop to iterate through the profile view -->
56                      {% for order in orders %}
57                          <tr>
58                              <td>
```

```
profile.html ×
profiles > templates > profiles > profile.html > ...
59                                  <!-- passes the order number through thte url -->
60                                  <a href="{% url 'order_history' order.order_number %}"
61                                  title="{{ order.order_number }}">
62                                      <!-- limits order number to six characthers -->
63                                      {{ order.order_number|truncatechars:6 }}
64                                  </a>
65                              </td>
66                              <!-- order date  -->
67                              <td>{{ order.date }}</td>
68                              <td>
69                                  <ul class="list-unstyled">
70                                      <!-- unordered list for size name and quanitity -->
71                                      {% for item in order.lineitems.all %}
72                                          <li class="small">
73                                              {% if item.product.has_sizes %}
74                                                  Size {{ item.product.size|upper }}
75                                              {% endif %}{{ item.product.name }} x{{ item.quantity }}
76                                          </li>
77                                      {% endfor %}
78                                  </ul>
79                              </td>
80                              <!-- grand total -->
81                              <td>${{ order.grand_total }}</td>
82                          </tr>
83                      {% endfor %}
84                  </tbody>
85              </table>
86          </div>
87      </div>
88  </div>
89  {% endblock %}
90  <!-- javascript to render stripe -->
91  {% block postloadjs %}
```

{% block content %}

Overlay and Container:

- To produce a backdrop effect, a div with the class "overlay" is used as an overlay.
- A "container" div is used to wrap the content on the page is this Bootstrap container.

Profile Heading:

- The profile heading is shown in a row with a single column.
- For visual separation, a horizontal line (hr>) is put above and below the heading.
- The <h2> element is used to display the heading itself, together with the class logo-font and the mb-4 margin.

Delivery Information:

- The Bootstrap class col-12 col-lg-6 is used to display the user delivery information in a column.
- "Default Delivery Information" is displayed in a paragraph with the class text muted.
- To amend the delivery information, a form is offered.
- Using the syntax % url 'profile'%, the form's action directs users to the URL with the name "profile."
- The data is safely sent via the POST method by the form.
- To prevent CSRF attacks, a CSRF token (% csrf_token%) is introduced.
- For improved styling, the form fields are presented using the form|crispy template filter.

Update Information Button:

- An "Update Information" button with the classes btn, btn-black, rounded-0, and text-uppercase is floated to the right of the column.

Order History :

- Another column utilizing the Bootstrap class col-12 col-lg-6 displays the user order history.
- "Order History" is displayed in a paragraph with the class text-muted.

Order History Table:

- With the help of the Bootstrap classes table, table-sm, and table-borderless, the order history is displayed in a table.
- The "Order Number," "Date," "Items," and "Order Total" columns are included in the table header.

Loop through Orders:

- Loop through Orders The context variable orders, which contains a list of order objects, is iterated through using a for-loop.
- A table row (tr) is produced for each order to display the order details.

Order Number Link:

- A link (a>) with the order number as a parameter is shown next to the order number and directs users to the 'order_history' URL.
- The full order number is displayed as a tooltip using the title attribute.
- The truncatechars template filter reduces the order number to six characters.

Order Number Link:

- In separate table fields, the order total, a list of the items, and the order date are all shown.

Conditional Size Display:

- To determine whether a product has sizes, a conditional statement is used (%if item.product.has_sizes%).
- If a product comes in different sizes, the size is listed first, then the product name and quantity.

Grand total:

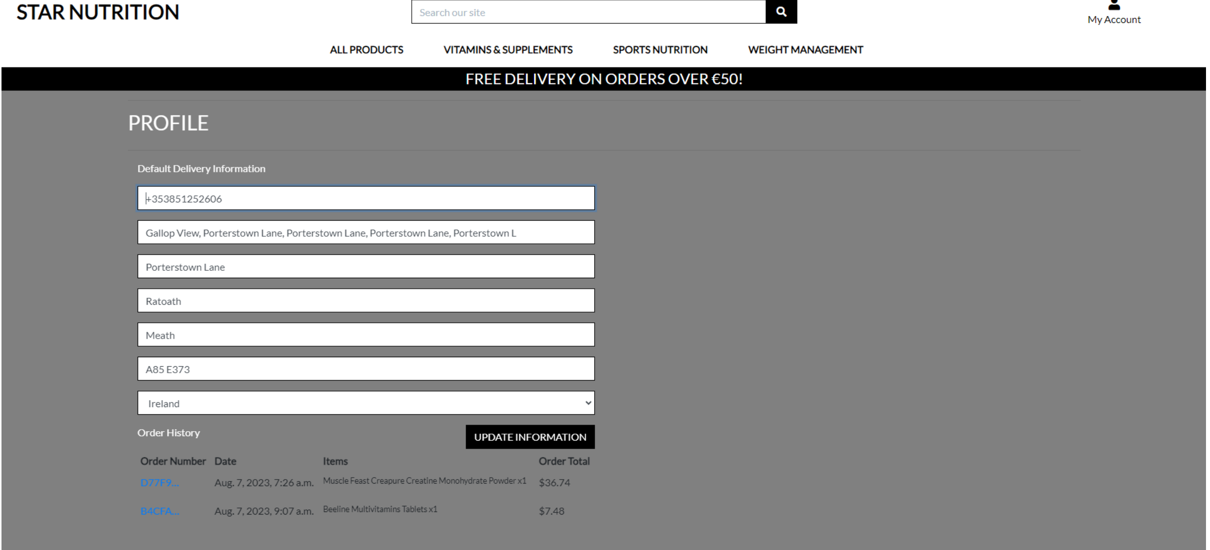
- The final column displays the overall sum for each order.

```
<!-- javascript to render stripe -->
{% block postloadjs %}
    {{ block.super }}
    <!-- template filter json_script to render the public key -->
    {{ stripe_public_key|json_script:"id_stripe_public_key" }}
    {{ client_secret|json_script:"id_client_secret" }}
    <!-- link to js file for the template -->
    <script src="{% static 'checkout/js/stripe_elements.js' %}"></script>
{% endblock %}
{% block postloadjs %}
    {{ block.super }}
    <!-- link to js file in static folder -->
    <script type="text/javascript" src="{% static 'profiles/js/countryfield.js' %}"></script>
{% endblock %}
```

{% block postloadjs %}

- For Stripe integration, postloadjs is used as the initial block. It contains utilizing the json_script template filter to generate the Stripe public key and client secret as JSON data.
- It is a reference to the stripe_elements.js JavaScript file in the static directory's checkout/js folder.
- Postloadjs_country, the second block, is for the Country Field. It contains a connection to the countryfield.js JavaScript file in the static directory's profiles/js folder.

Screenshot:



Shopping Cart:

The bag.html file extends the base.html file. The shopping bag page renders a shopping cart for users of the application to update their carts. The page was designed using HTML, and bootstrap CSS using Django template logic that renders the shopping bag details.

bag.html M ×

shoppingbag > templates > bag > bag.html > ...

```
1   {% extends "base.html" %}
2   {% load static %}
3   {% load bag_tool %}
4
5   {% block page_header %}
6   <div class="container header-container">
7       <div class="row">
8           <div class="col"></div>
9       </div>
10  </div>
11  {% endblock %}
12
```

{% extends "base.html" %}:

- This line instructs Django that the base.html template should be used as a basis for this template. It indicates that while this template can override particular blocks as necessary, it will use the layout and structure established in base.html.

{% load static %}:

- Similar to the base.html template, this template element loads static files.

{% block page_header %}:

- This child template replaces a block that is defined in the base.html template. The content from this child template is used to replace the page_header block's content in the base template. In this instance, it expands the header section to include a container with a row and column.

bag.html M

shoppingbag > templates > bag > bag.html > ...

```
{% block content %}
<!-- covers background image -->
<div class="overlay"></div>
<div class="container mb-2">
    <!-- row to contain shopping bag heading -->
    <div class="row">
        <div class="col">
            <hr>
            <h2 class="logo-font mb-4">Shopping Bag</h2>
            <hr>
        </div>
    </div>
    <!-- second row to contain a table showing back contents -->
    <div class="row">
        <div class="col">
            <!-- django logic to render table  -->
            <!-- if items in shopping bag table will be rendered -->
            {% if bag_items %}

            <div class="table-responsive rounded">
                <!-- table to show items in the bag taken from bootstrap docs -->
                <table class="table table-sm table-borderless">
                    <thead class="text-black">
                        <tr>
                            <!-- table headers for rendering products in shopping bag -->
                            <th scope="col">Product Description</th>
                            <th scope="col"></th>
                            <th scope="col">Price</th>
                            <th scope="col">Quantity</th>
                            <th scope="col">Subtotal</th>
                        </tr>
                    </thead>
                    <!-- iteration for each item in the shopping bag and return row for each item -->
                    {% for item in bag_items %}
                    <tr>
                        <!-- row for product image -->
                        <td class="p-3 w-25">
                            <img class="img-fluid rounded" src="{{ item.product.image.url }}">
                        </td>
                        <!-- row for product name -->
                        <td class="py-3">
                            <p class="my-0"><strong>{{ item.product.name}}</strong></p>
                            <p class="my-0"><strong>Size:
```

Ln 172, Col 21   Spaces: 4

bag.html M

shoppingbag > templates > bag > bag.html > ...

```
                                </strong>{% if item.product.has_sizes %}{{ item.size|upper }}{% else %}N/A{% endif %}
                            </p>
                        </td>
                        <!-- row for product price -->
                        <td class="py-3">
                            <!-- loads the price argument as the first argument for register filter and quantity as the second -->
                            <p class="my-0">{{ item.product.price }}</p>
                        </td>
                        <!-- row for product Quantity -->
                        <td class="py-3 w-25">
                            <!-- Allows updating quanity from the shopping bag -->
                            <form class="form update-form" method="POST" action="{% url 'adjust_bag' item.item_id %}">
                                {% csrf_token %}
                                <!-- form group form the product details page -->
                                <div class="form-group">
                                    <div class="input-group">
                                        <!-- add - button for taking away from item quantity -->
                                        <div class="input-group-prepend">
                                            <!-- buttons for javascript -->
                                            <!-- updating product variables to item id form product.id -->
                                            <button class="decrement-qty btn btn-sm btn-black rounded-0"
                                                data-item_id="{{ item.item_id }}" id="decrement-qty_{{ item.item_id }}">
                                                <span>
                                                    <i class="fas fa-minus fa-sm"></i>
                                                </span>
                                            </button>
                                        </div>
                                        <input class="form-control form-control-sm qty_input" type="number"
                                            name="quantity" value="{{ item.quantity }}" min="1" max="99"
                                            data-item_id="{{ item.item_id }}" id="id_qty_{{ item.item_id }}">
                                        <!-- add button for + button for adding to the product quanity -->
                                        <div class="input-group-append">
                                            <!-- buttons for javascript -->
                                            <button class="increment-qty btn btn-sm btn-black rounded-0"
                                                data-item_id="{{ item.item_id }}" id="increment-qty_{{ item.item_id }}">
                                                <span>
                                                    <i class="fas fa-plus fa-sm"></i>
                                                </span>
                                            </button>
                                        </div>
                                        <!-- size to be submitted if product has size -->
                                        {% if item.product.has_sizes %}
                                        <input type="hidden" name="product_size" value="{{ item.size }}">
                                        {% endif %}
                                    </div>
```

```
bag.html M ×
shoppingbag > templates > bag > bag.html > ...
101                         </div>
102                     </form>
103                     <!-- update items in the bag -->
104                     <a class="update-link text-info">Update</a>
105                     <!-- remove items from the bag -->
106                     <a class="remove-item text-danger float-right" id="remove_{{ item.item_id }}">
107                         <small>Remove</small></a>
108 
109                 </td>
110                 <!-- row for prouct price -->
111                 <td class="py-3">
112                     <p class="my-0">${{ item.product.price | calc_subtotal:item.quantity }}</p>
113                 </td>
114             </tr>
115             {% endfor %}
116             <tr>
117                 <td colspan="5" class="pt-5 text-right">
118                     <!-- return bag total -->
119                     <h6><strong>Bag Total: €{{ total|floatformat:2 }}</strong></h6>
120                     <!-- returns delivery charge -->
121                     <h6>Delivery: €{{ delivery|floatformat:2 }}</h6>
122                     <!-- free delivery delta if needed to spend more for free delivery -->
123                     <h4 class="mt-4"><strong>Grand Total: €{{ grand_total|floatformat:2 }}</strong></h4>
124                     {% if free_delivery_delta > 0 %}
125                     <p class="mb-1 text-danger">
126                         You could get free delivery by spending just <strong>€{{ free_delivery_delta }}</strong>
127                         more!
128                     </p>
129                     {% endif %}
130                 </td>
131             </tr>
132             <tr>
133                 <!-- keep shopping section button -->
134                 <td colspan="5" class="text-right">
135                     <a href="{% url 'products' %}" class="btn btn-outline-black rounded-0 btn-lg">
136                         <span class="icon">
137                             <i class="fas fa-chevron-left"></i>
138                         </span>
139                         <span class="text-uppercase">Continue Shopping</span>
140                     </a>
141                     <!-- button to procced to checkout -->
142                     <a href="{% url 'checkout' %}" class="btn btn-black rounded-0 btn-lg">
143                         <span class="text-uppercase">Secure Checkout</span>
144                         <span class="icon">
```

```
143                         <span class="text-uppercase">Secure Checkout</span>
144                         <span class="icon">
145                             <i class="fas fa-lock"></i>
146                         </span>
147                     </a>
148                 </td>
149             </tr>
150 
151         </table>
152     </div>
153     {% else %}
154     <!-- if not render paragtaph to explain bag is empty -->
155     <p class="lead mb-5">Your bag is empty! Please select a product you would like before checking out</p>
156     <!-- links user back to the products page -->
157     <a href="{% url 'products' %}" class="btn btn-outline-black rounded-0 btn-lg">
158         <span class="icon">
159             <i class="fas fa-chevron-left"></i>
160         </span>
161         <span class="text-uppercase">Continue Shopping</span>
162     </a>
163     {% endif %}
164     </div>
165   </div>
166 </div>
167 
168 {% endblock %}
```

{% block content %}

Overlay and Container:

- To produce a backdrop effect, a div with the class "overlay" is used as an overlay.
- A "container" div is used to wrap the content on the page is this Bootstrap container.

Shopping Bag Heading:

- The header "Shopping Bag" is presented with a horizontal line (hr) above and below it.

- The h2> element is used to display the heading "Shopping Bag" together with the class's logo-font and mb-4.

Shopping Bag Table:

- The items in the shopping bag are put on a table for viewing.
- The classes table, table-sm, and table-borderless are all present in the table.

Product Image, Name, Price, and Quantity:

- The product name, image, price, and quantity are all listed separately in each row's table cell.
- The img-fluid and rounded classes are used to display the product image as a responsive, rounded image.
- Along with the product price and quantity, the display also includes the product name and size (if item has a size).

Quantity Adjustment Form:

- Changes to the quantity of each item in the shopping bag are made using a form of the class update-form.
- The action on the form sends a request to the URL 'adjust_bag' with the item ID as an argument.
- For security, a CSRF token (%csrf_token%) is used.

Quantity Adjustment Buttons:

- Using input group components from Bootstrap, buttons for decreasing and increasing the quantity are included.
- These buttons have been using JavaScript functions to change the quantity.

Subtotal and Total:

- Each item's subtotal is computed and shown in the table's last column.
- Below the table, you can see the grand amount, delivery fee, and bag total.

Continue Shopping and Secure Checkout Buttons:

- Users can choose from buttons to move on to the secure payment page or to keep shopping.

Empty Bag Message:

- A message stating that the bag is empty is displayed if the bag is empty (i.e., bag_items is empty).
- To take people back to the products page, a button is included.

```
{% endblock %}
{% block postloadjs %}
{{ block.super }}

<!-- javascript remove and add buttons -->
<script type="text/javascript">
    // Update quantity on click
    $('.update-link').click(function (e) {
        var form = $(this).prev('.update-form');
        form.submit();
    })

    // Remove item and reload on click
    $('.remove-item').click(function (e) {
        var csrfToken = "{{ csrf_token }}";
        var itemId = $(this).attr('id').split('remove_')[1];
        var size = $(this).data('product_size');
        var url = `/shoppingbag/remove/${itemId}/`;
        var data = {
            'csrfmiddlewaretoken': csrfToken,
            'product_size': size
        };

        $.post(url, data)
            .done(function () {
                location.reload();
            });
    })
</script>
{% endblock %}
```

{% block postloadjs %}

Update Quantity:

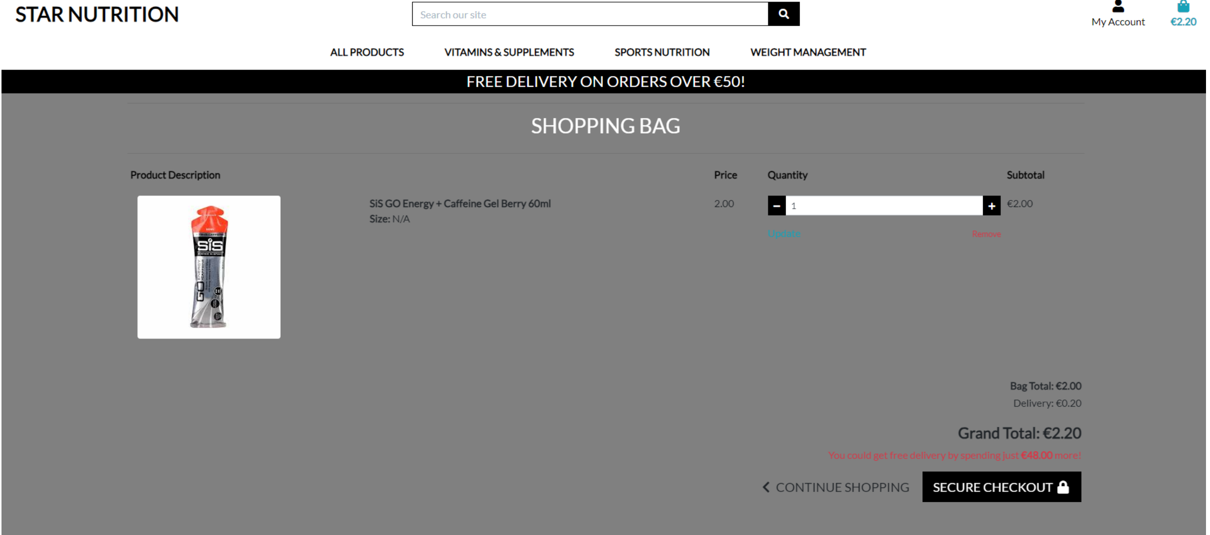
- When an element with the class update-link is clicked, the code creates an event listener for that event.
- The JavaScript function is triggered when the "Update" link is clicked.
- Using the prev() method, the function locates the previous element belonging to the class update-form (the quantity adjustment form).
- Once the form has been submitted, the quantity of the item in the shopping bag will be updated without refreshing the page.

Remove Item and Reload:

- When an element with the class remove-item is clicked, the code creates an event listener for that click.
- The JavaScript function is activated when the "Remove" button is clicked.
- The CSRF token (csrf_token) is first taken out of the template by the function and placed in the csrfToken variable.
- The split() and attr() functions are then used to retrieve the item ID from the clicked element's ID attribute.

- The size data is retrieved and saved in the size variable if the "Remove" button also includes a data-product_size property.
- The function then combines the item ID with the base URL /shopping bag/remove/ to create the URL for taking the item out of the bag.
- It generates a data object that includes the product_size and, if available, the CSRF token.
- The $.post() method is then used by the function to send a POST request with the data to the newly created URL.
- The function uses location.reload() to reload the page when the POST request is successful to update the shopping bag display without performing a full page refresh.

Screenshot:



Checkout:

The checkout.html file extends the base.html file. The checkout page renders a checkout page that gathers user data, shows the order summary, and enables customers to finish their transactions using Stripe as the payment processor. It handles order processing, payment verification, and order fulfilment. The page was designed using HTML, and bootstrap CSS using Django template logic that renders the shopping bag details.

```
ckout > templates > checkout > checkout.html > div.container > div.row > div.col > hr
{% extends "base.html" %}
{% load static %}
{% load bag_tool %}
<!-- adding css box for the template of the app -->
{% block extra_css %}
<link rel="stylesheet" href="{% static 'checkout/css/checkout.css' %}">
{% endblock %}

{% block page_header %}
<div class="container header-container">
    <div class="row">
        <div class="col"></div>
    </div>
</div>
{% endblock %}
```

{% extends "base.html" %}:

- This line instructs Django that the base.html template should be used as a basis for this template. It indicates that while this template can override particular blocks as necessary, it will use the layout and structure established in base.html.

{% load static %}:

- Similar to the base.html template, this template element loads static files.

{% block page_header %}:

- This child template replaces a block that is defined in the base.html template. The content from this child template is used to replace the page_header block's content in the base template. In this instance, it expands the header section to include a container with a row and column.

```
checkout.html
checkout > templates > checkout > checkout.html > div.container > div.row > div.col > hr
9   {% block page_header %}
10  <div class="container header-container">
11      <div class="row">
12          <div class="col"></div>
13      </div>
14  </div>
15  {% endblock %}
16
17  {% block content %}
18  <!-- covers background image -->
19  <div class="overlay"></div>
20  <div class="container">
21      <!-- row to contain shopping bag heading -->
22      <div class="row">
23          <div class="col">
24              <hr>
25              <h2 class="logo-font mb-4">Checkout</h2>
26              <hr>
27          </div>
28      </div>
29      <!-- second row to contain a table showing bag contents -->
30      <div class="row">
31          <!-- class for checkout to show up on the right of larger screens -->
32          <div class="col-12 col-lg-6 order-lg-last mb-5">
33              <p class="text-muted">Order Summary ({{ product_count }})</p>
34              <div class="row">
35                  <div class="col-7 offset-2">
36                      <p class="mb-1 mt-0 small text-muted">Item</p>
37                  </div>
38                  <div class="col-3 text-right">
39                      <p class="mb-1 mt-0 small text-muted">Subtotal</p>
40                  </div>
41              </div>
42          <!-- using context processor for the bag items to loop through each item -->
43          {% for item in bag_items %}
44          <div class="row">
45              <div class="col-2 mb-1">
46                  <!-- display iem image -->
47                  <a href="{% url 'product_detail' item.product.id %}">
48                      {% if item.product.image %}
49                      <img class="w-100" src="{{ item.product.image.url }}" alt="{{ product.name }}">
50                      {% else %}
51                      <img class="w-100" src="{{ MEDIA_URL }}noimage.png" alt="{{ product.name }}">
52                      {% endif %}
```

checkout.html

checkout > templates > checkout > checkout.html > div.container > div.row > div.col > hr

```
51                      <img class="w-100" src="{{ MEDIA_URL }}noimage.png" alt="{{ product.name }}">
52                      {% endif %}
53                  </a>
54              </div>
55              <div class="col-7">
56                  <!-- dispaly product name, quanity and the subttal -->
57                  <p class="my-0"><strong>{{ item.product.name }}</strong></p>
58                  <p class="my-0 small">Size: {% if item.product.has_sizes %}{{ item.size|upper }}{% else %}N/A{% endif %}
59                  </p>
60                  <p class="my-0 small text-muted">Qty: {{ item.quantity }}</p>
61              </div>
62              <div class="col-3 text-right">
63                  <p class="my-0 small text-muted">€{{ item.product.price | calc_subtotal:item.quantity }}</p>
64              </div>
65          </div>
66          {% endfor %}
67          <!-- order summary section  -->
68          <hr class="my-0">
69          <div class="row text-black text-right">
70              <div class="col-7 offset-2">
71                  <p class="my-0">Order Total:</p>
72                  <p class="my-0">Delivery:</p>
73                  <p class="my-0">Grand Total:</p>
74              </div>
75              <div class="col-3">
76                  <p class="my-0">${{ total | floatformat:2 }}</p>
77                  <p class="my-0">${{ delivery | floatformat:2 }}</p>
78                  <p class="my-0"><strong>${{ grand_total | floatformat:2 }}</strong></p>
79              </div>
80          </div>
81      </div>
82      <div class="col-12 col-lg-6">
83          <!-- instructions for users to fill oout forms -->
84          <p class="text-muted">Please fill out the form below to complete your order</p>
85          <!-- calling the checkout url for the POST method -->
86          <form action="{% url 'checkout' %}" method="POST" id="payment-form">
87              {% csrf_token %}
88              <!-- three field sets rendered for UI -->
89              <!-- fieldset for name and email -->
90              <fieldset class="rounded px-3 mb-5">
91                  <legend class="fieldset-label small text-black px-2 w-auto">Details</legend>
92                  <!-- using the cripsy form fields template for the users name and email-->
93                  {{ order_form.full_name | as_crispy_field }}
94                  {{ order_form.email | as_crispy_field }}
```

```
checkout.html ×
checkout > templates > checkout > checkout.html > div.container > div.row > div.col > hr
93                  {{ order_form.full_name | as_crispy_field }}
94                  {{ order_form.email | as_crispy_field }}
95              </fieldset>
96              <!-- fieldset two  -->
97              <fieldset class="rounded px-3 mb-5">
98                  <legend class="fieldset-label small text-black px-2 w-auto">Delivery</legend>
99                  <!-- using the cripsy form filed to display user delivery info -->
100                 {{ order_form.phone_number | as_crispy_field }}
101                 {{ order_form.street_address1 | as_crispy_field }}
102                 {{ order_form.street_address2 | as_crispy_field }}
103                 {{ order_form.town_or_city | as_crispy_field }}
104                 {{ order_form.county | as_crispy_field }}
105                 {{ order_form.postcode | as_crispy_field }}
106                 {{ order_form.country | as_crispy_field }}
107                 <!-- check box for user to save info to there profile -->
108                 <div class="form-check form-check-inline float-right mr-0">
109                     {% if user.is_authenticated %}
110                     <label class="form-check-label" for="id-save-info">Save this delivery information to my
111                         profile</label>
112                     <input class="form-check-input ml-2 mr-0" type="checkbox" id="id-save-info" name="save-info"
113                         checked>
114                     {% else %}
115                     <!-- if the user doesnt have an account a link to sign is provided -->
116                     <label class="form-check-label" for="id-save-info">
117                         <a class="text-info" href="{% url 'account_signup' %}">Create an account</a> or
118                         <a class="text-info" href="{% url 'account_login' %}">login</a> to save this information
119                     </label>
120                     {% endif %}
121                 </div>
122             </fieldset>
123             <!-- fieldset 3 for sstripe payments -->
124             <fieldset class="px-3">
125                 <legend class="fieldset-label small text-black px-2 w-auto">Payment</legend>
126                 <!-- A Stripe card element will go here -->
127                 <div class="mb-3" id="card-element"></div>
128                 <!-- Used to display form errors -->
129                 <div class="mb-3 text-danger" id="card-errors" role="alert"></div>
130                 <!-- Pass the client secret to the view so we can get the payment intent id -->
131                 <input type="hidden" value="{{ client_secret }}" name="client_secret">
132             </fieldset>
133             <!-- submit button  -->
134             <div class="submit-button text-right mt-5 mb-2">
135                 <!-- link back to shopping bag if user wants to buy another product -->
136                 <a href="{% url 'view_bag' %}" class="btn btn-outline-black rounded-0">
137                     <span class="icon">
```

```
checkout.html ×
checkout > templates > checkout > checkout.html > div.container > div.row > div.col > hr
138                         <i class="fas fa-chevron-left"></i>
139                     </span>
140                     <span class="font-weight-bold">Adjust Bag</span>
141                 </a>
142                 <!-- submit button to confrim purchase -->
143                 <button id="submit-button" class="btn btn-black rounded-0">
144                     <span class="font-weight-bold">Complete Order</span>
145                     <span class="icon">
146                         <i class="fas fa-lock"></i>
147                     </span>
148                 </button>
149                 <!-- lets the user know thier card will be charged -->
150                 <p class="small text-danger my-0">
151                     <span class="icon">
152                         <i class="fas fa-exclamation-circle"></i>
153                     </span>
154                     <span>Your card will be charged <strong>€{{ grand_total|floatformat:2 }}</strong></span>
155                 </p>
156             </div>
157         </form>
158     </div>
159 </div>
160 <!-- div overlay for when payment is been processed -->
161 <div id="loading-overlay">
162     <h1 class="text-light logo-font loading-spinner">
163         <span class="icon">
164             <i class="fas fa-3x fa-sync-alt fa-spin"></i>
165         </span>
166     </h1>
167 </div>
168 {% endblock %}
```

{% block content %}

Overlay and Container:

- To produce a backdrop effect, a div with the class "overlay" is used as an overlay.
- A "container" div is used to wrap the content on the page is this Bootstrap container.

Order Summary Section:

- Presents a table with a list of the goods in the shopping bag.
- It displays the product image (if accessible), name, size (if appropriate), quantity, and subtotal for each item.
- The values for "Order Total," "Delivery," and "Grand Total" are also displayed at the bottom.

Form Section:

- Contains a form for gathering user data for payment and shipping information.
- The user's full name and email are fields in fieldset 1's "Details" section.
- The user's phone number, street address, city, county, postal code, and country are all included in Fieldset 2's "Delivery" fields. Additionally, if the user is authenticated, a checkbox to save delivery information to their profile is present. If not, it offers opportunities to register or log in.
- The Stripe card element will be displayed here to manage the payment. Fieldset 3: "Payment" In a hidden input area for the payment intent, the client's secret value is also there.

Submit Button:

- Provides the user with a link to return their shopping bag if they need to alter it.
- The form is submitted for order completion and payment processing by clicking the "Complete Order" button.
- Additionally, there is a notification letting the consumer know that the "Grand Total" will be charged to their card.

Integration with Stripe:

Stripe Card Element:

- The Stripe card element will be displayed in the div with the id="card-element" tag. Users can safely enter their credit card information with this component.

Overlay for Payment Processing:

- When the payment is being processed, the div with the ID "loading-overlay" is used to display an overlay with a spinning icon. Users can see this as a good visual cue that the payment is active.

```
</div>
{% endblock %}
<!-- javascript to render stripe -->
{% block postloadjs %}
    {{ block.super }}
    <!-- template filter json_script to render the public key -->
    {{ stripe_public_key|json_script:"id_stripe_public_key" }}
    {{ client_secret|json_script:"id_client_secret" }}
    <!-- link to js file for the template -->
    <script src="{% static 'checkout/js/stripe_elements.js' %}"></script>
{% endblock %}
```

{% block postloadjs %}

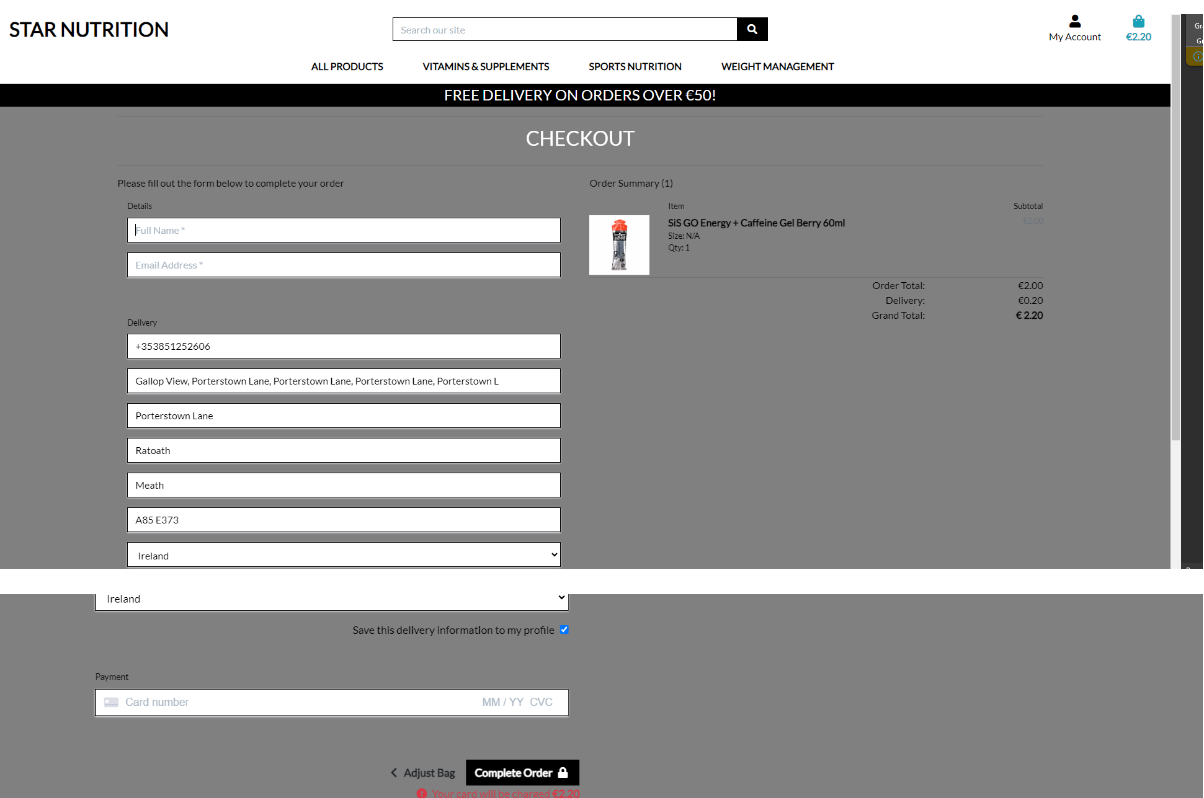
Public Key and Client Secret:

- The json_script template filter is used to generate the stripe_public_key and client_secret variables as JSON scripts. To interface with Stripe on the client side, these variables are required.

Stripe JavaScript File:

- The script> tag contains a link to the stripe_elements.js JavaScript file, which is in charge of managing the configuration of the Stripe card element and form submission.

Screenshot:



Checkout Success:

The checkout_sucess.html file extends the base.html file. The checkout success page renders the order information and provides a link for users to go back to their profile. The page was designed using HTML, and bootstrap CSS using Django template logic that renders the shopping bag details.

```
checkout_success.html ×
checkout > templates > checkout > checkout_success.html > div.container > div.row > div.col-12.col-lg-7 > div.order-c
1   {% extends "base.html" %}
2   {% load static %}
3
4   <!-- adding css box for the template of the app -->
5   {% block extra_css %}
6   <link rel="stylesheet" href="{% static 'checkout/css/checkout.css' %}">
7   {% endblock %}
8
9   {% block page_header %}
10  <div class="container header-container">
11      <div class="row">
12          <div class="col"></div>
13      </div>
14  </div>
15  {% endblock %}
16
17  {% block content %}
```

{% extends "base.html" %}:

- This line instructs Django that the base.html template should be used as a basis for this template. It indicates that while this template can override particular blocks as necessary, it will use the layout and structure established in base.html.

{% load static %}:

- Similar to the base.html template, this template element loads static files.

{% block page_header %}:

- This child template replaces a block that is defined in the base.html template. The content from this child template is used to replace the page_header block's content in the base template. In this instance, it expands the header section to include a container with a row and column.

checkout_success.html

checkout > templates > checkout > checkout_success.html > div.container > div.row > div.col-12.col-lg-7 > div.order-confirmation-wrapper.p-2.border > div.row > div.col-12.col-md-8

```
17  {% block content %}
18  <!-- covers background image -->
19  <div class="overlay"></div>
20  <div class="container">
21      <!-- row to contain checkout success heading -->
22      <div class="row">
23          <div class="col">
24              <hr>
25              <h2 class="logo-font mb-4">Checkout Success Thank You!!</h2>
26              <hr>
27              <p class="text-black">Your order information is below. A confirmation email will be sent to
28                  <strong>{{ order.email }}</strong>.</p>
29          </div>
30      </div>
31      <!-- second row to contain a table showing bag contents -->
32      <div class="row">
33          <div class="col-12 col-lg-7">
34              <!-- wrap order confirmation with a border -->
35              <div class="order-confirmation-wrapper p-2 border">
36                  <!-- section for order info-->
37                  <div class="row">
38                      <div class="col">
39                          <small class="text-muted">Order Information:</small>
40                      </div>
41                  </div>
42                  <!-- section for order number -->
43                  <div class="row">
44                      <div class="col-12 col-md-4">
45                          <!-- add order number text -->
46                          <p class="mb-0 text-black font-weight-bold">Order Number:</p>
47                      </div>
48                      <!-- add order number itself  -->
49                      <div class="col-12 col-md-8 text-md-right">
50                          <p class="mb-0">{{ order.order_number }}</p>
51                      </div>
52                  </div>
53                  <!-- section for order date-->
54                  <div class="row">
55                      <div class="col-12 col-md-4">
56                          <!-- order date text -->
57                          <p class="mb-0 text-black font-weight-bold">Order Date:</p>
58                      </div>
59                      <div class="col-12 col-md-8 text-md-right">
60                          <!-- actual order date -->
```

checkout_success.html

checkout > templates > checkout > checkout_success.html > div.container > div.row > div.col-12.col-lg-7 > div.order-confirmation-wrapper.p-2.border > div.row >

```
59                  <div class="col-12 col-md-8 text-md-right">
60                      <!-- actual order date -->
61                      <p class="mb-0">{{ order.date }}</p>
62                  </div>
63              </div>
64              <!-- section for order details-->
65              <div class="row">
66                  <div class="col">
67                      <small class="text-muted">Order Details:</small>
68                  </div>
69              </div>
70              <!-- django logic for loop for getting line items -->
71              {% for item in order.lineitems.all %}
72              <div class="row">
73                  <div class="col-12 col-md-4">
74                      <p class="small mb-0 text-black font-weight-bold">
75                          <!-- generate the name, size and price -->
76                          {{ item.product.name }}{% if item.product_size %} - Size
77                          {{ item.product.size|upper }}{% endif %}
78                      </p>
79                  </div>
80                  <div class="col-12 col-md-8 text-md-right">
81                      <!-- generate the item quanity and price  -->
82                      <p class="small mb-0">{{ item.quantity }} @ ${{ item.product.price }} each</p>
83                  </div>
84              </div>
85              {% endfor %}
86              <!-- uses the order model -->
87              <div class="row">
88                  <div class="col">
89                      <small class="text-muted">Delivering To:</small>
90                  </div>
91              </div>
92              <!-- gets full name -->
93              <div class="row">
94                  <div class="col-12 col-md-4">
95                      <p class="mb-0 text-black font-weight-bold">Full Name</p>
96                  </div>
97                  <div class="col-12 col-md-8 text-md-right">
98                      <p class="mb-0">{{ order.full_name }}</p>
99                  </div>
100             </div>
101             <!-- gets street address 1 -->
102             <div class="row">
```

checkout_success.html

checkout > templates > checkout > checkout_success.html > div.container > div.row > div.col-12.col-lg-7 > div.order-confirmation-wrapper.p-2.border > div.row > div.col-12.col-md-8.text-md-right

```
101             <!-- gets street address 1 -->
102             <div class="row">
103                 <div class="col-12 col-md-4">
104                     <p class="mb-0 text-black font-weight-bold">Address 1</p>
105                 </div>
106                 <div class="col-12 col-md-8 text-md-right">
107                     <p class="mb-0">{{ order.street_address1 }}</p>
108                 </div>
109             </div>
110             <!-- gets street address 2 -->
111             {% if order.street_address2 %}
112             <div class="row">
113                 <div class="col-12 col-md-4">
114                     <p class="mb-0 text-black font-weight-bold">Address 2</p>
115                 </div>
116                 <div class="col-12 col-md-8 text-md-right">
117                     <p class="mb-0">{{ order.street_address1 }}</p>
118                 </div>
119             </div>
120             {% endif %}
121             <!-- gets county -->
122             {% if order.county %}
123             <div class="row">
124                 <div class="col-12 col-md-4">
125                     <p class="mb-0 text-black font-weight-bold">County</p>
126                 </div>
127                 <div class="col-12 col-md-8 text-md-right">
128                     <p class="mb-0">{{ order.county }}</p>
129                 </div>
130             </div>
131             {% endif %}
132             <!-- gets the town or city -->
133             <div class="row">
134                 <div class="col-12 col-md-4">
135                     <p class="mb-0 text-black font-weight-bold">Town or City</p>
136                 </div>
137                 <div class="col-12 col-md-8 text-md-right">
138                     <p class="mb-0">{{ order.town_or_city }}</p>
139                 </div>
140             </div>
141             <!-- gets the post code -->
142             {% if order.postcode %}
143             <div class="row">
144                 <div class="col-12 col-md-4">
```

checkout_success.html ×

checkout > templates > checkout > checkout_success.html > div.container > div.row > div.col-12.col-lg-7 > div.order-confirmation-wrapper.p-2.border > div.row > div.col-12.col-md-8.text-md-right

```
143                 <div class="row">
144                     <div class="col-12 col-md-4">
145                         <p class="mb-0 text-black font-weight-bold">Postal Code</p>
146                     </div>
147                     <div class="col-12 col-md-8 text-md-right">
148                         <p class="mb-0">{{ order.postcode }}</p>
149                     </div>
150                 </div>
151                 {% endif %}
152                 <!-- gets the county -->
153                 <div class="row">
154                     <div class="col-12 col-md-4">
155                         <p class="mb-0 text-black font-weight-bold">Country</p>
156                     </div>
157                     <div class="col-12 col-md-8 text-md-right">
158                         <p class="mb-0">{{ order.country }}</p>
159                     </div>
160                 </div>
161                 <!-- gets the phone number -->
162                 <div class="row">
163                     <div class="col-12 col-md-4">
164                         <p class="mb-0 text-black font-weight-bold">Phone Number</p>
165                     </div>
166                     <div class="col-12 col-md-8 text-md-right">
167                         <p class="mb-0">{{ order.phone_number }}</p>
168                     </div>
169                 </div>
170                 <!-- gets the billing information -->
171                 <div class="row">
172                     <div class="col">
173                         <small class="text-muted">Billing Info:</small>
174                     </div>
175                 </div>
176                 <!-- gets the order total -->
177                 <div class="row">
178                     <div class="col-12 col-md-4">
179                         <p class="mb-0 text-black font-weight-bold">Order Total</p>
180                     </div>
181                     <div class="col-12 col-md-8 text-md-right">
182                         <p class="mb-0">{{ order.order_total }}</p>
183                     </div>
184                 </div>
185                 <!-- gets the delivery cost -->
186                 <div class="row">
```

checkout_success.html ×

checkout > templates > checkout > checkout_success.html > div.container > div.row > div.col-12.col-lg-7 > div.order-confirmation-wrapper.p-2.border > div.row > div.col-12.col-md-8.text-m

```
185                 <!-- gets the delivery cost -->
186                 <div class="row">
187                     <div class="col-12 col-md-4">
188                         <p class="mb-0 text-black font-weight-bold">Delivery</p>
189                     </div>
190                     <div class="col-12 col-md-8 text-md-right">
191                         <p class="mb-0">{{ order.delivery_cost }}</p>
192                     </div>
193                 </div>
194                 <!-- gets the grand total -->
195                 <div class="row">
196                     <div class="col-12 col-md-4">
197                         <p class="mb-0 text-black font-weight-bold">Grand Total</p>
198                     </div>
199                     <div class="col-12 col-md-8 text-md-right">
200                         <p class="mb-0">{{ order.grand_total }}</p>
201                     </div>
202                 </div>
203             </div>
204         </div>
205     </div>
206     <div class="row">
207         <!-- back to profile button -->
208         <div class="col-12 col-lg-7 text-right">
209             <a href="{% url 'profile' %}" class="btn btn-black rounded-0 my-2">
210                 <span class="icon mr-2">
211                     <i class="fas fa-angle-left"></i>
212                 </span>
213                 <span class="text-uppercase">Back to Profile</span>
214             </a>
215         </div>
216     </div>
217 </div>
218 {% endblock %}
219 <!-- javascript to render stripe -->
```

{% block content %}

Overlay and Container:

- To produce a backdrop effect, a div with the class "overlay" is used as an overlay.
- A "container" div is used to wrap the content on the page is this Bootstrap container.

Order Information Table:

- The table offers a structured format for showing order details such as the order number, order date, order details (including product names, sizes, and prices), delivery address, billing information, order total, shipping cost, and overall total.
- Django template tags and filters are used to obtain data from the order model and populate the content.

Back to Profile Button:

- The "Back to Profile" button is used when clicked it directs users to their profile page.
- To maintain consistency with the overall design, the button is styled using Bootstrap classes.

```
<!-- javascript to render stripe -->
{% block postloadjs %}
{{ block.super }}
<!-- template filter json_script to render the public key -->
{{ stripe_public_key|json_script:"id_stripe_public_key" }}
{{ client_secret|json_script:"id_client_secret" }}
<!-- link to js file for the template -->
<script src="{% static 'checkout/js/stripe_elements.js' %}"></script>
{% endblock %}
```

{% block postloadjs %}

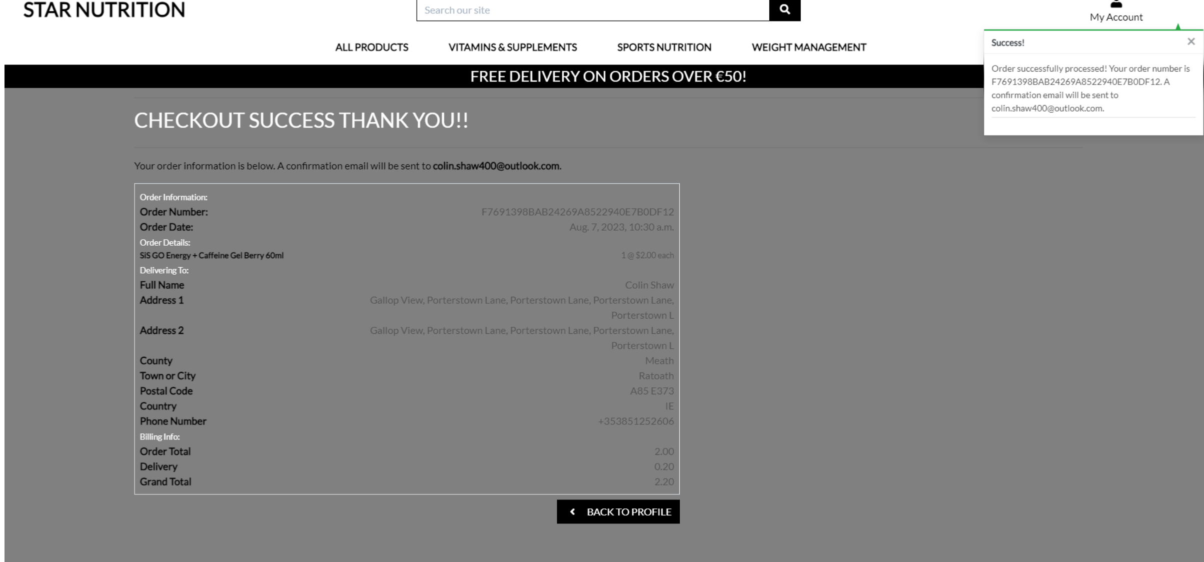
Public Key and Client Secret:

- The json_script template filter is used to generate the stripe_public_key and client_secret variables as JSON scripts. To interface with Stripe on the client side, these variables are required.

Stripe JavaScript File:

- The script> tag contains a link to the stripe_elements.js JavaScript file, which is in charge of managing the configuration of the Stripe card element and form submission.

Screenshot:



Toasts:

Success

The toast_success.html code is a Django-template-based HTML template. It is a custom template that shows a success message.

```
toast_success.html
templates > includes > toasts > toast_success.html > div.toast.custom-toast.rounded-0.border-top-0 > div.toast-body.bg-white > div.row
1  <!-- toast templates taken from bootstrap to give a success message -->
2  <div class="toast custom-toast rounded-0 border-top-0" data-autohide="false">
3      <div class="arrow-up arrow-success"></div>
4      <div class="w-100 toast-capper bg-success"></div>
5      <div class="toast-header bg-white text-dark">
6          <strong class="mr-auto">Success!</strong>
7          <button type="button" class="ml-2 mb-1 close text-dark" data-dismiss="toast" aria-label="Close">
8              <span aria-hidden="true">&times;</span>
9          </button>
10     </div>
11     <!-- preview of shopping bag in succes toast -->
12     <div class="toast-body bg-white">
13         <div class="row">
14             <div class="col">
15                 {{ message }}
16                 <hr class="mt-1 mb-3">
17             </div>
18         </div>
19         <!-- grand total template -->
20         {% if grand_total and not on_profile_page %}
21             <p class="logo-font bg-white text-black py-1">Your Bag ({{ product_count }})</p>
22             <div class="bag-notification-wrapper">
23                 {% for item in bag_items %}
24                     <div class="row">
25                         <div class="col-3 my-1">
26                             <img class="w-100" src="{{ item.product.image.url }}">
27                         </div>
28                         <div class="col-9">
29                             <p class="my-0"><strong>{{ item.product.name }}</strong></p>
30                             <p class="my-0 small">Size: {% if item.product.has_sizes %}{{ item.size|upper }}{% else %}N/A{% endif %}</p>
31                             <p class="my-0 small text-muted">Qty: {{ item.quantity }}</p>
32                         </div>
33                     </div>
34                 {% endfor %}
35             </div>
36             <div class="row">
37                 <div class="col">
38                     <!-- delivery delata template -->
39                     <strong><p class="mt-3 mb-1 text-black">
40                         Total{% if free_delivery_delta > 0 %} (Exc. delivery){% endif %}:
41                         <span class="float-right">€{{ total|floatformat:2 }}</span>
42                     </p></strong>
43                     {% if free_delivery_delta > 0 %}
44                         <p class="mb-0 p-2 bg-warning shadow-sm text-black text-center">
```

```
toast_success.html
templates > includes > toasts > toast_success.html > div.toast.custom-toast.rounded-0.border-top-0 > div.toast-body.bg-white > div.row
41                              <span class="float-right">€{{ total|floatformat:2 }}</span>
42                          </p></strong>
43                          {% if free_delivery_delta > 0 %}
44                              <p class="mb-0 p-2 bg-warning shadow-sm text-black text-center">
45                                  Spend <strong>€{{ free_delivery_delta }}</strong> more to get free next day delivery!
46                              </p>
47                          {% endif %}
48                          <a href="{% url 'view_bag' %}" class="btn btn-black btn-block rounded-0">
49                              <span class="text-uppercase">Go To Secure Checkout</span>
50                              <span class="icon">
51                                  <i class="fas fa-lock"></i>
52                              </span>
53                          </a>
54                      </div>
55                  </div>
56              {% endif %}
57          </div>
58      </div>
```

Toast Container:

- The toast message container is represented by the outermost div with the class "toast".

Arrow:

- There is an arrow element with the classes "arrow-up" and "arrow-success" that points upward.

Toast Header:

- The toast's header features the phrase "Success!" shown as a strong element (strong>), as well as an "x" (×) close button for dismissing the toast. White writing on a dark background makes up the header.

Toast Body:

- The "toast-body" div, which is styled with a white background, contains the bulk of the toast's content.

Message:

- The "message" variable is a placeholder for dynamic material that will be shown as the toast's main message.

Grand Total:

- The conditional block is based on the grand_total and on_profile_page variables included inside the toast. If these criteria are satisfied, the content about the shopping bag and the total will be displayed.

Shopping Bag Content:

- If the criteria are satisfied, the products in the shopping bag will be displayed. Each item is shown as a row that includes the product's name, size (if available), and number along with an image of the item.

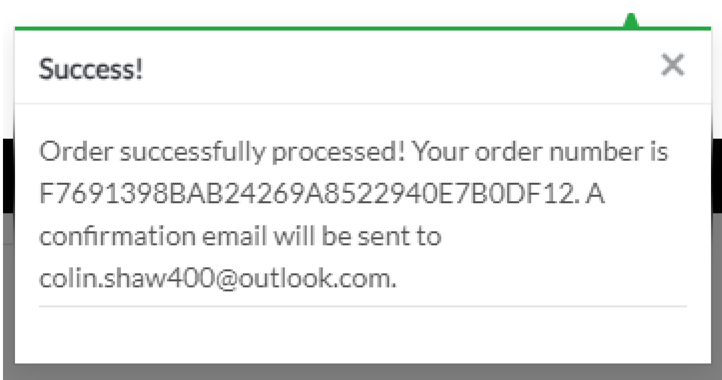
Grand Total:

- It shows the total cost (in euros) of the products in the shopping bag and, if the free_delivery_delta value is more than 0, it prompts the customer to spend additional money to qualify for free next-day delivery.

Go To Secure Checkout Button:

- The option to access the secure checkout page is labelled "Go To Secure Checkout." "Go To Secure Checkout" is written there, and a locked padlock with a Font Awesome icon serves as the icon.

Screenshot:



Error

The toast_error.html code is a Django-template-based HTML template. It is a custom template that shows an error message.

```
toasts_error.html
templates > includes > toasts > toasts_error.html > div.toast.custom-toast.rounded-0.border-top-0
1  <!-- toast error message taken from bootstrap -->
2  <div class="toast custom-toast rounded-0 border-top-0" data-autohide="false">
3      <div class="arrow-up arrow-danger"></div>
4      <div class="w-100 toast-capper bg-danger"></div>
5      <div class="toast-header bg-white text-dark">
6          <strong class="mr-auto">Error!</strong>
7          <button type="button" class="ml-2 mb-1 close text-dark" data-dismiss="toast" aria-label="Close">
8              <span aria-hidden="true">&times;</span>
9          </button>
10     </div>
11     <div class="toast-body bg-white">
12         {{ message }}
13     </div>
14 </div>
```

Toast Container:

- The toast message container is represented by the outermost div with the class "toast".

Arrow:

- Like the previous template, this one also features a distinctive upward-pointing arrow element. This time, it has the classes "arrow-up" and "arrow-danger," which indicate that it is styled for an error message.

Toast Header:

- The toast's header has an "Error!" title that is displayed as a strong element (strong>) and an "x" (×) close button that can be used to dismiss the toast. White writing on a dark background makes up the header.

Toast Body:

- The "toast-body" div, which is styled with a white background, contains the bulk of the toast's content.

Message:

- The message variable is a placeholder for a dynamic text that will be shown as the toast's primary error message.

Information:

The toast_info.html code is a Django-template-based HTML template. It is a custom template that shows an information message.

```
toasts_info.html ×
templates > includes > toasts > toasts_info.html > div.toast.custom-toast.rounded-0.border-top-0 > div.toast-body.bg-white
1   # toast info message
2   <div class="toast custom-toast rounded-0 border-top-0" data-autohide="false">
3       <div class="arrow-up arrow-info"></div>
4       <div class="w-100 toast-capper bg-info"></div>
5       <div class="toast-header bg-white text-dark">
6           <strong class="mr-auto">Alert!</strong>
7           <button type="button" class="ml-2 mb-1 close text-dark" data-dismiss="toast" aria-label="Close">
8               <span aria-hidden="true">&times;</span>
9           </button>
10      </div>
11      <div class="toast-body bg-white">
12          {{ message }}
13      </div>
14  </div>
```

Toast Container:

- The toast message container is represented by the outermost div with the class "toast".

Arrow:

- Like the earlier templates, this one also has a distinctive upward-pointing arrow element. This time it has the classes "arrow-up" and "arrow-info," it is styled for an alert or informational message.

Toast Header:

- The toast's header includes the word "Alert!" shown as a strong element (strong>), as well as an "x" (×) close button for dismissing the toast. White writing on a dark background makes up the header.

Toast Body:

- The "toast-body" div, which is styled with a white background, contains the bulk of the toast's content.
- Message: The message variable is a placeholder for dynamic material that will be shown as the toast's primary information or alert message.

Warning:

The toast_warning.html code is a Django-template-based HTML template. It is a custom template that shows a warning message.

toasts_warning.html

templates > includes > toasts > toasts_warning.html > div.toast.custom-toast.rounded-0.border-top-0

```
<!-- toast warning message  -->
<div class="toast custom-toast rounded-0 border-top-0" data-autohide="false">
    <div class="arrow-up arrow-warning"></div>
    <div class="w-100 toast-capper bg-warning"></div>
    <div class="toast-header bg-white text-dark">
        <strong class="mr-auto">Warning!</strong>
        <button type="button" class="ml-2 mb-1 close text-dark" data-dismiss="toast" aria-label="Close">
            <span aria-hidden="true">&times;</span>
        </button>
    </div>
    <div class="toast-body bg-white">
        {{ message }}
    </div>
```

Toast Container:

- The toast message container is represented by the outermost div with the class "toast".

Arrow:

- Similar to the earlier templates, this one has a distinctive upward-pointing arrow element. This time, it has the classes "arrow-up" and "arrow-warning," which denote that it is styled for a warning message.

Toast Header:

- The toast's header includes the word "Warning!" shown as a strong element (strong>), as well as an "x" (×) close button for dismissing the toast. White writing on a dark background makes up the header.

Toast Body:

- The "toast-body" div, which is styled with a white background, contains the bulk of the toast's content.

Message:

- The message variable is a placeholder for dynamic material that will be shown as the toast's primary warning message.

Fixtures:

Fixtures are used in JSON, XML, or YAML file formats. They serve as an image of all the data in your application, including model instances and the data they are linked to. Fixtures can be used to populate data for a variety of purposes, including setting up sample material, pre-loading reference data, and creating default user accounts. The folder is found in the application of the product in the fixtures folder.

In this application, they are used to load the product categories and product details using a JSON file.

Categories:

The categories.json are represented as a list of product categories in JSON format together with the corresponding data. A primary key (pk), a model type (model), and a set of fields with the category name (name) and a friendly name (friendly_name) are all present for each category.

To load the categories the command line command is used. “python3 manage.py load data categories”.

Here is an example of the first category loaded below.

[{"pk": 1, "model": "products.category", "fields": {"name": "vitamins", "friendly_name": "Vitamins"}}]

Products:

The products.json is a JSON representation that contains data about the application's products. A primary key (pk), a model type (model), and some fields with information on the product—including its SKU, name, description, price, category, rating, and image—are included in each entry for a given product. this helps preload the database with the product information.

To load the products the command line command is used. “python3 manage.py loaddata products”

Here is an example of the first product loaded below.


```
    "pk": 1,
    "model": "products.product",
    "fields": {
        "SKU": "p000000001",
        "name": "Beeline Multivitamins Tablets",
        "description": "Beeline Multivitamin Tablets contain our most advanced vitamin and mineral formula to provide comprehensive nutritional support. Suitable for vegetarians, the supplement supports health, vitality, wellbeing and energy levels.",
        "price": 6.80,
        "category": 1,
        "rating": 4.4,
        "image": "multi.jpg"
    }
},
```

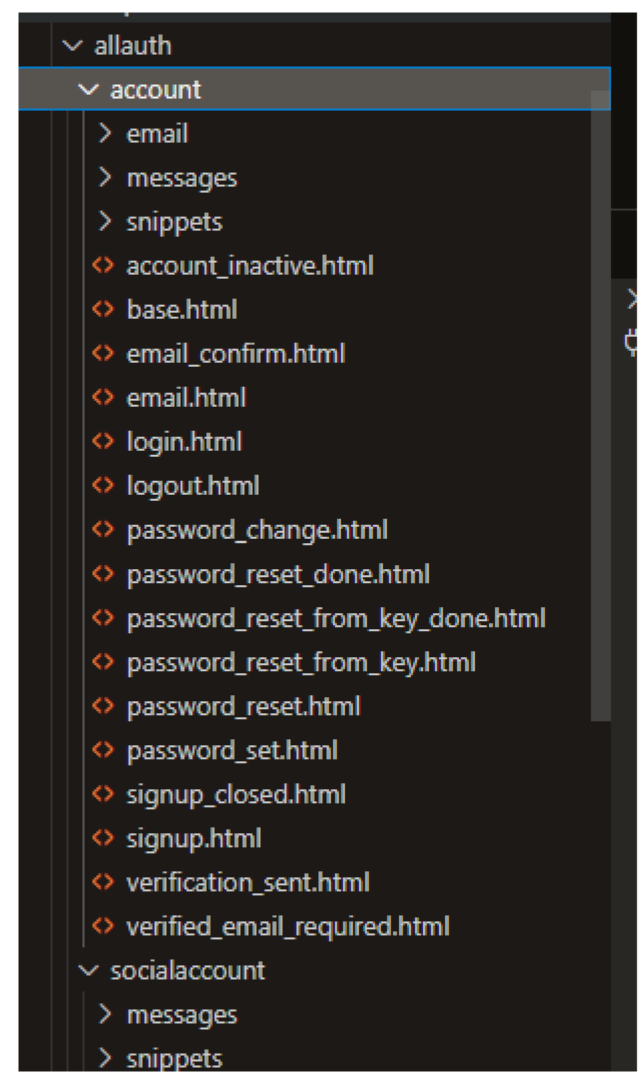

Allauth:

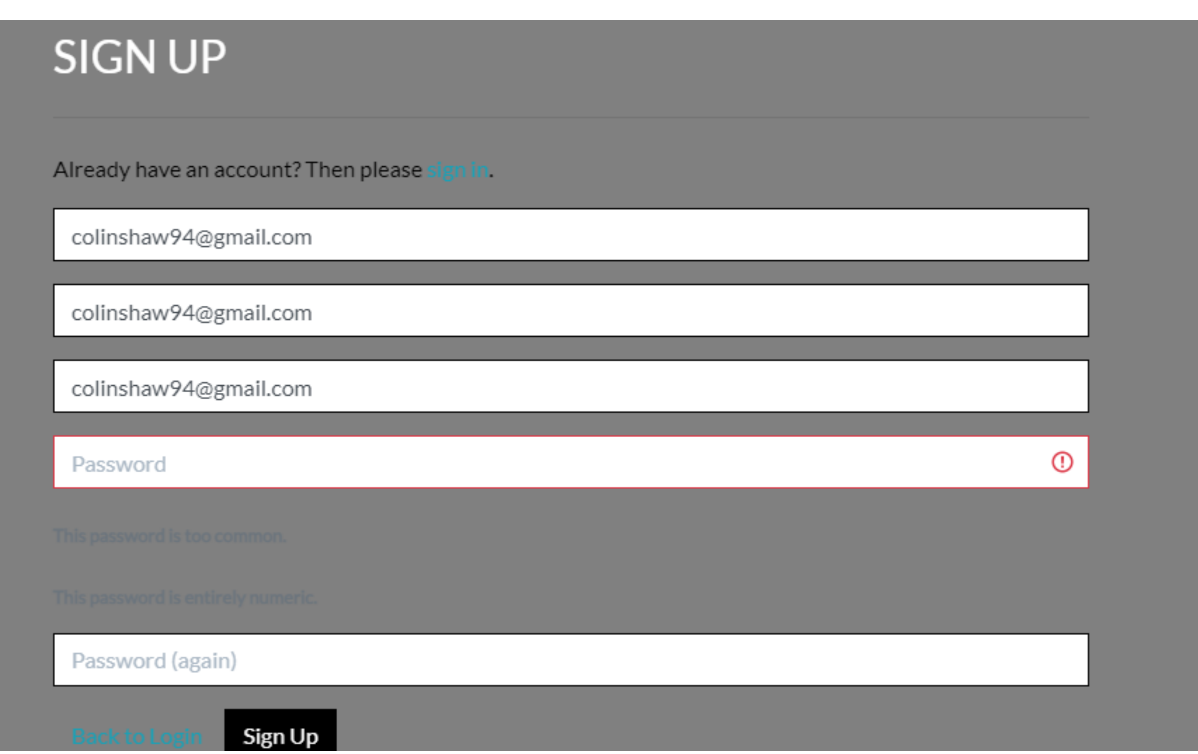
The well-known Django package "allauth" offers a selection of templates and views for authentication and account management functions, such as user registration, login, password reset, and other capabilities. Allauth version 0.41.0 is used as it worked best for the application and was easiest to customise the templates.

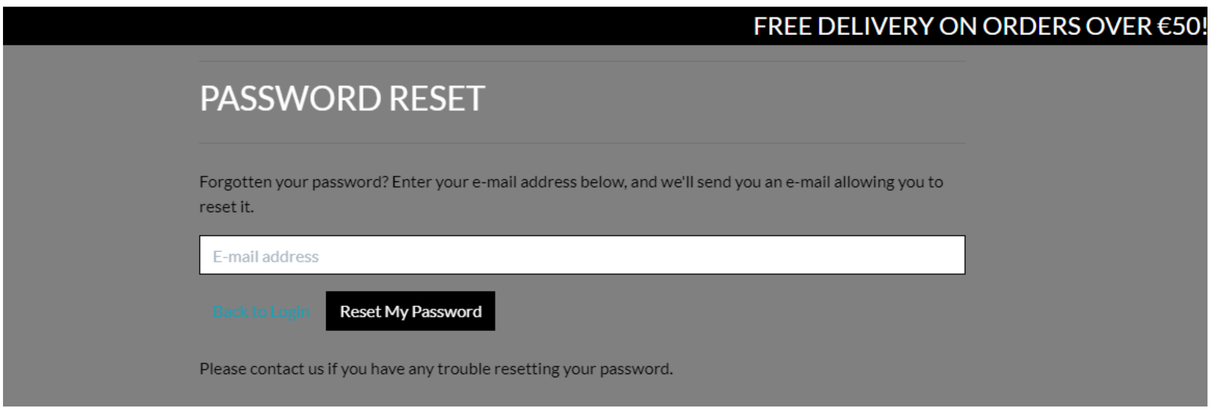
Once Django was installed I was able to copy the templates from the allauth package to the alluath template directory in the application. Once in the directory, the command used was.

```
cp -r ../.pip-modules/lib/python3.8/site-packages/allauth/templates/* ./templates/allauth/
```

Once the template the directory looked like this and then allauth templates could be rendered and looked like the screenshots below.


allauth
account
email
messages
snippets
account_inactive.html
base.html
email_confirm.html
email.html
login.html
logout.html
password_change.html
password_reset_done.html
password_reset_from_key_done.html
password_reset_from_key.html
password_reset.html
password_set.html
signup_closed.html
signup.html
verification_sent.html
verified_email_required.html
socialaccount
messages
snippets


SIGN UP
Already have an account? Then please sign in.
colinshaw94@gmail.com
colinshaw94@gmail.com
colinshaw94@gmail.com
Password
This password is too common.
This password is entirely numeric.
Password (again)
Back to Login
Sign Up



Static CSS:

A static css file called base.css is used to style the application.

Global Styles:

- In the first section, the body's height and the background image settings are among the global styles that are applied to the html and body components.

Font Awesome Icon Styling:

- Font Awesome icons are shown using the defined styles for the .icon class. The alignment and icon dimensions are set using flexbox attributes.

Logo Styling:

- To apply text modifications, such as making the text uppercase, use the.logo-font class. It probably appears in headlines or logos.

Button Styling:

- Several button styles, including.shop-now-button,.btn-black, and.btn-outline-black, are defined. These styles have various backgrounds and text colors for the states (hover, active, and focus), respectively.

Category Badges Styling:

- The styles for category badges are defined here. The background color turns to black and the text color to white when the badges are hovered over.

Back to Top Button:

- Back-to-top button styling for the.btt-button class. It is a tiny button that consistently appears in the bottom-right corner of the display.

Input Placeholder Styling

- Custom styling for the placeholder of input elements with the name "q" is used. A particular color will be used for the placeholder text.

Bootstrap Toasts Styling:

- Using classes like.message-container,.custom-toast,.toast-capper,.arrow-up, and various.arrow-color classes, Bootstrap toasts are styled to display different coloured arrows for different toast types (primary, secondary, success, danger, warning, info, light, dark).

Bag Notification Styling:

- Styles for the wrapper of the bag notification, which shows the contents of the shopping bag.

Media Queries:

- Media queries are used to apply various styles depending on the size of the screen (e.g., changing the container's width on extremely big screens, or modifying the padding on smaller devices).

Allauth Form Styling:

- Form styling for the "allauth" package, which handles user registration and authentication. Input fields, buttons, links, and labels are all customised.

Fixed Top Navbar:

- For medium-sized and bigger screens, styles for a fixed-top navigation bar are available.

Header Styling:

- Depending on the size of the screen, different padding and font sizes are adjusted for the top part of the website.

Profile:

This profiles.css styles the order history and profile update form area of an application.

```
# profiles.css ×
profiles > static > css > # profiles.css > #id_default_country option:first-child
1  .order-history {
2      max-height: 416px; /* height of profile form + submit button */
3      overflow-y: auto;
4  }
5
6  #profile-update-form .form-control {
7      color: #000;
8  }
9
10 #profile-update-form input::placeholder {
11     color: #aab7c4;
12 }
13
14 #id_default_country,
15 #id_default_country option:not(:first-child) {
16     color: #000;
17 }
18
19 #id_default_country option:first-child {
20     color: #aab7c4;
21 }
```

Order History Styling:

- A container that displays the order history is given the.order-history class. When the content surpasses the container's maximum height of 416 pixels, vertical scrolling is enabled.

Profile Update Form Styling:

- The input elements in the profile update form are targeted by the #profile-update-form.form-control selector. It changes the input elements' text color to black.

Input Placeholder Styling for Profile Update Form:

- The placeholders contained within the input elements of the profile update form are the focus of the #profile-update-form input::placeholder selector. It changes the color of the placeholder text to a pale grey (#aab7c4).

Country Dropdown Styling:

- The default country dropdown element is the focus of the #id_default_country selector. It changes the dropdown menu's text color to black.

Country Dropdown Placeholder Styling:

- The first choice in the dropdown, which is typically a placeholder, is not targeted by the #id_default_country option:not(:first-child) selector. It changes these options' text colors to black.

Country Dropdown Placeholder Styling:

- The first option of the dropdown, which is typically a placeholder like "Select a Country," is the focus of the #id_default_country option:first-child selector. This option's text is set to the same light grey (#aab7c4) as the input placeholders' text.

Checkout:

This checkout.css styles the overlays, as well as some additional styling for the integration of Stripe elements and form controls.

```
# checkout.css ×
checkout > static > checkout > css > # checkout.css > ...
1   /* stripe css from stripe docs */
2   .StripeElement,
3   .stripe-style-input {
4     box-sizing: border-box;
5     height: 40px;
6     padding: 10px 12px;
7     border: 1px solid transparent;
8     border-radius: 0px;
9     background-color: white;
10    box-shadow: 0 1px 3px 0 #e6ebf1;
11    -webkit-transition: box-shadow 150ms ease;
12    transition: box-shadow 150ms ease;
13  }
14
15  .StripeElement--focus,
16  .stripe-style-input:focus,
17  .stripe-style-input:active {
18    box-shadow: 0 1px 3px 0 #cfd7df;
19  }
20
21  .StripeElement--webkit-autofill {
22    background-color: #fefde5 !important;
23  }
24
25  .stripe-style-input::placeholder {
26      color: #aab7c4;
27  }
28
29
```

Stripe Element and Form Control Styling:

- Styling for Stripe elements and form controls, which are utilized in the processing of payments. The appearance of Stripe components and input fields is specified by the.StripeElement and.stripe-style-input classes. They consist of the attributes for box-sizing, height, padding, border, border-radius, background color, and box-shadow. When an element or input field is in focus, the StripeElement--focus and.stripe-style-input:focus classes specify the appearance.

Stripe Element Autocomplete Styling:

- When the Stripe element holds data that was autocompleted, the StripeElement--webkit-autofill class changes the background color.

Placeholder Styling for Input Fields:

- The input fields' placeholder text color is set by the.stripe-style-input::placeholder class.

```
0   /* fieldset label adjustment */
1   .fieldset-label {
2       position: relative;
3       right: .5rem;
4   }
5
6   /* all form controls to have black text and border */
7   #payment-form .form-control,
8   #card-element {
9       color: #000;
0       border: 1px solid #000;
1   }
```

Fieldset Label Adjustment:

- The fieldset label's location is changed by the.fieldset-label class.

Form Control Styling for Payment Form:

- The #payment-form has form control styling for the payment form. The text color and border for form controls used in the payment form are configured using the form-control and #card-element selectors.

```
41  }
42  /* styling for country fields */
43  select,
44  select option {
45      color: #000000;
46  }
47
48  select:invalid,
49  select option[value=""] {
50      color: #aab7c4 !important;
51  }
52
```

Styling for Country Fields:

- Styles for nation selection fields, such as the color of the options and how to handle incorrect selections, are available.

```css
/* over lay css for checkout  */
#loading-overlay {
    display: none;
    position: fixed;
    top: 0;
    left: 0;
    width: 100%;
    height: 100%;
    background: rgba(23, 162, 184, .85);
    z-index: 9999;
}

.loading-spinner {
    display: flex;
    align-items: center;
    justify-content: center;
    margin: 0;
    height: 100%;
}
```

Overlay Styling for Checkout:

- Styles for the overlay that will be utilized during the checkout process. The position and look of the overlay, when it is shown, are set by the #loading-overlay selector. The loading spinner is centred inside the overlay using the.loading-spinner class.

Application level file(star_app):

Settings.py:

```python
14  # Allows us to import the os module
15  import os
16  # Build paths inside the project like this: BASE_DIR / 'subdir'.
17  BASE_DIR = Path(__file__).resolve().parent.parent
18
19
20  # Quick-start development settings - unsuitable for production
21  # See https://docs.djangoproject.com/en/3.2/howto/deployment/checklist/
22
23  # SECURITY WARNING: keep the secret key used in production secret!
24  SECRET_KEY = 'django-insecure-+^mc1sld#+35r&c2ep++gsnmdur8teu)qtds5=l1r(_t2mrsno'
25
26  # SECURITY WARNING: don't run with debug turned on in production!
27  DEBUG = True
28
29  ALLOWED_HOSTS = ['8000-colinshaw1-starnutritio-wfudkbs11uc.ws-eu102.gitpod.io']
30
31  CSRF_TRUSTED_ORIGINS = ['https://8000-colinshaw1-starnutritio-wfudkbs11uc.ws-eu102.gitpod.io']
32
33  # Application definition
34
35  INSTALLED_APPS = [
36      'django.contrib.admin',
37      'django.contrib.auth',
38      'django.contrib.contenttypes',
39      'django.contrib.sessions',
40      'django.contrib.messages',
41      'django.contrib.staticfiles',
42      'django.contrib.sites',
43      'homepage',
44      'products',
45      'shoppingbag',
46      'checkout',
47      'profiles',
48      # adds allauth to installed apps
49      'allauth',
50      'allauth.account',
51      'allauth.socialaccount',
52
53      # other apps
54      'crispy_forms',
55  ]
```

The settings.py file is crucial for configuration and acts as the central file for the entire project and is located in the applications main directory.

DEBUG:

- The DEBUG setting, which is meant for development environments, is set to True. When failures happen, it enables you to view comprehensive error pages with stack traces.

ALLOWED_HOSTS:

- A list of hostnames that the application is permitted to serve is contained in the ALLOWED_HOSTS option. It only permits queries from the IP address '8000-colinshaw1-starnutritio-wfudkbs11uc.ws-eu102.gitpod.io'.

CSRF_TRUSTED_ORIGINS:

- Cross-Site Request Forgery (CSRF) protection is provided by CSRF_TRUSTED_ORIGINS, which specifies trusted origins. These origins' requests won't need CSRF tokens.

INSTALLED_APPS:

- Lists each Django application that has been installed in the project. The applications consist of 'Django.contrib' apps, third-party apps like 'allauth' for authentication and 'crispy_forms' for form styling, as well as custom apps like 'homepage', 'products','shoppingbag', 'checkout', and 'profiles'.

```
56
57  MIDDLEWARE = [
58      'django.middleware.security.SecurityMiddleware',
59      'django.contrib.sessions.middleware.SessionMiddleware',
60      'django.middleware.common.CommonMiddleware',
61      'django.middleware.csrf.CsrfViewMiddleware',
62      'django.contrib.auth.middleware.AuthenticationMiddleware',
63      'django.contrib.messages.middleware.MessageMiddleware',
64      'django.middleware.clickjacking.XFrameOptionsMiddleware',
65  ]
66
67  ROOT_URLCONF = 'star_app.urls'
68
69  #telling dajngo which template will use crispy forms
70  CRISPY_TEMPLATE_PACK = 'bootstrap4'
71
72  TEMPLATES = [
73      {
74          'BACKEND': 'django.template.backends.django.DjangoTemplates',
75          'DIRS': [
76              os.path.join(BASE_DIR, 'templates'),
77              os.path.join(BASE_DIR, 'templates', 'allauth'),
78          ],
79          'APP_DIRS': True,
80          'OPTIONS': {
81              'context_processors': [
82                  'django.template.context_processors.debug',
83                  'django.template.context_processors.request',
84                  'django.contrib.auth.context_processors.auth',
85                  'django.contrib.messages.context_processors.messages',
86                  'django.template.context_processors.media',
87                  # adding shopping bag context process, so bag contents will be available in alll tmeplates
88                  'shoppingbag.contexts.bag_contents',
89              ],
90              # tells django what templates to use for crispy forms
91              'builtins': [
92                  'crispy_forms.templatetags.crispy_forms_tags',
93                  'crispy_forms.templatetags.crispy_forms_field',
94              ]
95          },
96      },
97  ]
98
```

MIDDLEWARE:

- Defines the collection of middleware classes that the application will use. Each request and answer trigger a different sequence of middleware executions.

ROOT_URLCONF:

- Points to the project's URL configuration, which contains the routing rules.

TEMPLATES:

- Specifies how HTML template configuration should be rendered. Context processors, template directories, and other options are included.

DATABASES:

- Describes the settings of the database. In this instance, a file called "db.sqlite3" houses an SQLite database.

```
# message storage required for git pod
MESSAGE_STORAGE = 'django.contrib.messages.storage.session.SessionStorage'


AUTHENTICATION_BACKENDS = (
    # Needed to login by username in Django admin, regardless of `allauth`
    'django.contrib.auth.backends.ModelBackend',

    # `allauth` specific authentication methods, such as login by e-mail
    'allauth.account.auth_backends.AuthenticationBackend',
)

SITE_ID = 1

EMAIL_BACKEND = 'django.core.mail.backends.console.EmailBackend'

ACCOUNT_AUTHENTICATION_METHOD = 'username_email'
ACCOUNT_EMAIL_REQUIRED = True
ACCOUNT_EMAIL_VERIFICATION = 'mandatory'
ACCOUNT_SIGNUP_EMAIL_ENTER_TWICE = True
ACCOUNT_USERNAME_MIN_LENGTH = 4
LOGIN_URL = '/accounts/login/'
LOGIN_REDIRECT_URL = '/'
WSGI_APPLICATION = 'star_app.wsgi.application'


# Database
# https://docs.djangoproject.com/en/3.2/ref/settings/#databases

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        # had to make database a string to stop PosixPath error
        'NAME': str(BASE_DIR / 'db.sqlite3'),
    }
}
```

AUTHENTICATION_BACKENDS:

- Lists the user authentication backends in use, including the 'allauth' and default Django backends.

SITE_ID:

- The ID of the current Django site is SITE_ID. The setting is 1.

EMAIL_BACKEND:

- Sets the email backend's settings so that emails can be sent. 'Django.core.mail.backends.console.EmailBackend' is used in this instance to display emails in the console while developing.

ACCOUNT_* settings:

- Settings for user accounts and authentication using the 'allauth' package are contained in the ACCOUNT_* file.

```
155
156    # Internationalization
157    # https://docs.djangoproject.com/en/3.2/topics/i18n/
158
159    LANGUAGE_CODE = 'en-us'
160
161    TIME_ZONE = 'UTC'
162
163    USE_I18N = True
164
165    USE_L10N = True
166
167    USE_TZ = True
168
```

LANGUAGE_CODE:

- Defines the application's default language code.

TIME_ZONE:

Defines the application's default time zone.

```
169
170    # Static files (CSS, JavaScript, Images)
171    # https://docs.djangoproject.com/en/3.2/howto/static-files/
172
173    STATIC_URL = '/static/'
174    #tells django where all static files are located. file has to be a tuple(())
175    STATICFILES_DIRS = (os.path.join(BASE_DIR, 'static'),)
176
177    #for all uploaded media files
178    MEDIA_URL = '/media/'
179    MEDIA_ROOT = os.path.join(BASE_DIR, 'media')
180
181    # Default primary key field type
182    # https://docs.djangoproject.com/en/3.2/ref/settings/#default-auto-field
183
184    DEFAULT_AUTO_FIELD = 'django.db.models.BigAutoField'
185
186    # add variables to calculate delivery costs
187    FREE_DELIVERY_THRESHOLD = 50
188    STANDARD_DELIVERY_PERCENTAGE = 8
189
190    # Stripe
191    FREE_DELIVERY_THRESHOLD = 50
192    STANDARD_DELIVERY_PERCENTAGE = 10
193    # sets curencey to euro
194    STRIPE_CURRENCY = 'eur'
195    # setting stripe public key
196    STRIPE_PUBLIC_KEY = os.getenv('STRIPE_PUBLIC_KEY', '')
197    # setting the secret stripe key
198    STRIPE_SECRET_KEY = os.getenv('STRIPE_SECRET_KEY', '')
199    # setting stripe webhook secret key
200    STRIPE_WH_SECRET = os.getenv('STRIPE_WH_SECRET', '')
```

STATIC_URL and STATICFILES_DIRS:

- Settings for managing static files including CSS, JavaScript, and pictures are STATIC_URL and STATICFILES_DIRS.

MEDIA_URL and MEDIA_ROOT:

- Settings for managing uploaded media files include MEDIA_URL and MEDIA_ROOT.

DEFAULT_AUTO_FIELD:

- The default primary key field type for models is specified by DEFAULT_AUTO_FIELD.

FREE_DELIVERY_THRESHOLD and STANDARD_DELIVERY_PERCENTAGE:

- Variables used to estimate shipping costs for the online store include FREE_DELIVERY_THRESHOLD and STANDARD_DELIVERY_PERCENTAGE.
- STRIPE_* settings:
- Configuration options for integrating Stripe payment processing are found in the STRIPE_* settings.

URLS:

```
urls.py ×
star_app > urls.py > {} admin
1  from django.contrib import admin
2  from django.urls import path, include
3  # add imports for static files
4  from django.conf import settings
5  from django.conf.urls.static import static
6
7
8  urlpatterns = [
9      path('admin/', admin.site.urls),
10     # # adding alluath urls to project level file
11     path('accounts/', include('allauth.urls')),
12     #adding homepage urls to project level file
13     path('', include('homepage.urls')),
14     # adding products urls to project level file
15     path('products/', include('products.urls')),
16     # adding shoppingbag urls to project level file
17     path('shoppingbag/', include('shoppingbag.urls')),
18     # adding checkout urls to project level file
19     path('checkout/', include('checkout.urls')),
20     # adding profile urls to project level file
21     path('profile/', include('profiles.urls')),
22 ] + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
23 #using static function to add url to media urls
24
```

The main application urls.py file is an essential part of the application for routing the systems URLS. It works by mapping incoming requests to the appropriate views and endpoints within the application. Each URL pattern is associated with a view function that handles a request and returns the appropriate response.

Import Django Modules:

- The admin site and other built-in Django applications are located in the Django.contrib module.

Django.urls:

- The Django URL manipulation tools are contained in this module.

Import Static Files Related Modules:

- Access to Django's settings variables is made possible through the django.conf.settings module.

django.conf.urls.static:

- During development, you can use this module to provide static resources like CSS, JavaScript, and pictures.

URLS:

- /admin/: The Django admin interface.
- /accounts/: URLs related to user authentication and registration using allauth.
- homepage/: The homepage URLs.
- /products/: URLs related to products.
- /shopping bag/: URLs related to the shopping bag or cart.
- /checkout/: URLs related to the checkout process.
- /profile/: URLs related to user profiles.

- + static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
- While MEDIA_ROOT is the absolute filesystem path to the directory holding media files, MEDIA_URL is defined in Django's settings.py file and points to the URL where media files are served.

Homepage:

Models:

The homepage has no models file as it does not have any use for one.

URLS:

```
urls.py
homepage > urls.py > {} admin
1  from django.contrib import admin
2  from django.urls import path
3  from . import views
4
5  urlpatterns = [
6      # add url for homepage
7      path('', views.index, name='homepage'),
8  ]
```

In the urls.py located on the homepage application. The index view function will be run when a user accesses the app's root URL, returning the proper homepage response. The logic to render the homepage's content will be found in the view function index, which is anticipated to be defined in the homepage's views.py module.

Import Django Modules:

- The admin site and other built-in Django applications are located in the django.contrib module.

django.urls:

- The Django URL manipulation tools are contained in this module.

from . import views:

- The views are imported from the current app using the from. import command. The view functions that process various HTTP requests and produce the necessary responses are contained in the views module.

path('', views.index, name='homepage'):

- The URL pattern for the app's home page. Django calls the index view function from the app's views.py module when a user accesses the root URL, and the view function processes the request and returns an HTTP response to render the homepage content.

Views:

```
views.py ×
homepage > views.py > render
1  from django.shortcuts import render
2
3  # Create your views here.
4
5  def index(request):
6      """ A view to return the index.html page"""
7
8      return render(request, 'home/index.html')
```

Import:

- Render: from django.shortcuts import importing the render method from Django's shortcuts module. Rendering templates and returning HTTP replies are done by this function.

def index(request):

- This is a view function called index. It is in charge of responding to HTTP requests from the homepage URL.

return render(request, 'home/index.html'):

returns:

- This line renders the index.html template and returns it as an HTTP response using the render function.

request:

- The user's HTTP request object includes details about the browser they used, any session data, and any supplied data (GET/POST parameters).

'home/index.html':

- This is the relative path to the index file within the templates directory of the homepage app. It instructs Django to use the templates folder of the app's home directory's index.html template.

Admin:

- The admin.py file is not called on for the homepage.

Products:

Models:

The models specify how the database tables that store categories and products should be structured. The Product model represents specific products with their features, such as name, description, and price, while the Category model is used to categorise products.

Category Model:

```
# Create your models here.
# category model to seperate the products into categories
class Category(models.Model):
    # add meta class to change the spelling of categorys in admin to categories
    class Meta:
        verbose_name_plural = 'Categories'
    # name of the catgory to help make the name more code readable for views etc
    name = models.CharField(max_length=254)
    # friendly name of the category that will show up on the application
    friendly_name = models.CharField(max_length=254, null=True, blank=True)

    # string to take in the category name and return the name
    def __str__(self):
        return self.name

    # model method to returnt he friendly name if needed
    def get_friendly_name(self):
        return self.friendly_name
```

- The model's Meta class is used to alter the model's behaviour via the Django admin interface. In this instance, it transforms the model's plural name "Categories" into the correct "Categories."
- It serves as a representation of the product category and is utilised in the application to classify products.
- A character field (CharField) with a 254 character maximum is used for the name field. The name of the category is stored.
- A character field (CharField) with a 254 character limit is also used for the friendly_name field. It keeps a name for the category that may be seen in the user-friendly programme.
- __str__() method:
- The string representation of the model instance, which is the name of the category, is returned by the __str__() method.
- get_friendly_name() Method:
- A custom model method named get_friendly_name() can be used to access the friendly_name property of a Category instance. If the category's friendly_name has been set, it will be returned. It will return None if not.

Product Model:

```python
# product model
class Product(models.Model):
    # foreign key to call upon the category model, if category is deleted so if any models use it null will be set
    category = models.ForeignKey('Category', null=True, blank=True, on_delete=models.SET_NULL)
    sku = models.CharField(max_length=254, null=True, blank=True)
    name = models.CharField(max_length=254)
    description = models.TextField()
    # add sizes model so can selct sizes on certain products
    has_sizes = models.BooleanField(default=False, null=True, blank=True)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    rating = models.DecimalField(max_digits=6, decimal_places=2, null=True, blank=True)
    image = models.ImageField(null=True, blank=True)

    # string method to return the product name same as category model
    def __str__(self):
        return self.name
```

- A product is represented by it in the application.
- A foreign key (ForeignKey) that connects the product to a category is the category field. It enables the product to be connected to a particular category, and if that category is eliminated, the on_delete=models function is activated.Instead of removing the product, the SET_NULL option guarantees that the category will be set to null in the product.
- A character field (CharField) with a 254 character maximum is the SKU field. It is used as an identifier number for the products.json file in the fixtures folder.
- A character field (CharField) with a 254 character maximum is used for the name field. The product's name is kept there.
- A longer description of the product may be entered in the description box, which is a text field (TextField).
- Has_sizes are a boolean field (BooleanField) that specifies whether or not the product is available in various sizes. The product has sizes if True; else, it does not.
- A maximum of 6 digits and 2 decimal places can be stored in the price field (DecimalField), which is a decimal field.
- With a maximum of 6 digits and 2 decimal places, the rating field (DecimalField) stores the rating of the product. It is not necessary and can be null or blank.
- An image for the product may be uploaded using the image field. It is not necessary and can be null or blank.
- __str__() method:
- The string representation of the model instance, which is the name of the category, is returned by the __str__() method.

URLS:

```python
urls.py
products > urls.py > {} path
from django.urls import path
from . import views

# view to call products
urlpatterns = [
    path('', views.products_view, name='products'),
    # on product id return products details
    path('<product_id>', views.product_detail, name='product_detail'),
]
```

Import Django Modules:

- django.urls: The Django URL manipulation tools are contained in this module.

from . import views:

- The views are imported from the current app using the from. import command. The view functions that process various HTTP requests and produce the necessary responses are contained in the views module.

path('', views.products_view, name='products'):

- The products_view view function is associated with this address. It represents the root URL of the app The products_view function will be used when a user accesses this URL to display the products page with a list of the available products. 'products' is the term given to the URL pattern, which can be used to reverse the URL for the products page.

path('<product_id>', views.product_detail, name='product_detail'):

- This is used to display information about a specific product. A URL parameter called "product_id" stores a product's unique identifier (ID). The function will then retrieve the relevant product details and show them on the product detail page. The 'product_detail' URL pattern can be used to reverse the URL to reveal information about any particular product.

Views:

The product view functions to manage and show products and their details in the application. The 'products.html' template is in charge of showing a list of items, while the 'product_detail.html' template is in charge of showing specific product details.

Imports:

```
views.py ×
products > views.py > render
1  from django.shortcuts import render, get_object_or_404
2  from django.contrib import messages
3  from django.db.models import Q
4  from django.db.models.functions import Lower
5  from .models import Product, Category
```

render:

- Render is a Django shorthand for generating template renderings and HTTP response backends.

get_object_or_404:

- The function get_object_or_404 retrieves an object from the database using its primary key (pk) and raises a 404 page not found error if the object is missing.

messages:

- A Django module is used to show users messages (such as success, error, and warning) while processing requests.

Q:

- A class from Django's query objects that enables the creation of sophisticated database queries utilising OR and AND logic.

Lower:

- A database function from Django that is used for queries' case-insensitive sorting.
- Models imported from the app's models.py file for the products and categories.
- Product and Category: These models correspond to product and category database tables.

products_view(request):

```
views.py ×
products > views.py > render
9   def products_view(request):
10      """ A view to show all products, including sorting and search queries """
11
12      # return products form databse
13      products = Product.objects.all()
14      #stops an error wehn loading products page without a search term
15      query = None
16      #query for returning selected products by categories
17      categories = None
18      # query for returning sproducts in sort query
19      sort = None
20      # query for returning sproducts in direction query
21      direction = None
22
23      # checking if requests. GET exists
24      if request.GET:
25
26          # if statment to sort by pricing, rating or category in a certain direction
27          if 'sort' in request.GET:
28              sortkey = request.GET['sort']
29              sort = sortkey
30              # annotate to allow case sensitive sorting
31              if sortkey == 'name':
32                  sortkey = 'lower_name'
33                  products = products.annotate(lower_name=Lower('name'))
34
35              # condtional to see if sort key is equal to category and if is return name
36              if sortkey == 'category':
37                  sortkey = 'category__name'
38
39              if 'direction' in request.GET:
40                  direction = request.GET['direction']
41                  if direction == 'desc':
42                      sortkey = f'-{sortkey}'
43              products = products.order_by(sortkey)
44
45          # if statment for selected categories to be returned
46          if 'category' in request.GET:
47              categories = request.GET['category'].split(',')
48              #filter products down to only the categories that are in the list
49              products = products.filter(category__name__in=categories)
50              # filter all categories
51              categories = Category.objects.filter(name__in=categories)
52
```

views.py ×

products > views.py > render

```
51              categories = Category.objects.filter(name__in=categories)
52
53          # if q is in request.GET equal query
54          if 'q' in request.GET:
55              query = request.GET['q']
56              # if q is in request.GET equal error message
57              if not query:
58                  messages.error(request, "Please check your search query and try agian?")
59                  return redirect(reversal('products'))
60
61              queries = Q(name__icontains=query) | Q(description__icontains=query)
62              # passing to filter message to filter products
63              products = products.filter(queries)
64      # fortmainting sort
65      current_sorting = f'{sort}_{direction}'
66
67      context = {
68          'products': products,
69          # query added to context
70          'search_term' : query,
71          # category objects
72          'current_categories':categories,
73          #sorting object
74          'current_sorting': current_sorting,
75      }
76      #return on products.html page
77      return render(request, 'products/products.html', context)
78
```

- This view function manages the products page, which showcases all of the products. It has features for categorising and sorting products according to search terms and chosen categories.
- The code initially uses the database to retrieve all products using Product.objects.all()
- It determines whether any query parameters are present in the GET request and then takes the appropriate steps following those parameters. For instance, it may filter products based on certain categories and search terms, and it can sort products by name, category, cost, or rating.
- The products variable holds the sorted and filtered products.
- The function additionally receives and processes information from the GET request regarding search queries (q), selected categories (category), sorting, and direction (sort and direction).
- The filtered products, search terms, chosen categories, and current sorting options are used to generate the context dictionary.
- The 'products.html' template is then rendered with the context and returned as the HTTP response by the function.

product_detail(request, product_id):

```python
79
80  def product_detail(request, product_id):
81      """ A view to single product details """
82
83      # taking in project id and getting object 404
84      product = get_object_or_404(Product, pk=product_id)
85
86      context = {
87          'product': product,
88      }
89      # return on product details
90      return render(request, 'products/product_detail.html', context)
```

- The product detail page, which displays comprehensive details about a single product, is handled by this view function.
- The get_object_or_404 function is used by the function to retrieve the specific product with the specified ID from the database. The function accepts the argument product_id.
- If the specified product ID is invalid, a 404 page not found message is returned.
- The product variable holds the product object.
- The product information is used to generate the context dictionary.
- The 'product_detail.html' template is then rendered with the context and returned as the HTTP response by the function.

Admin:

The Django admin is a great feature for customising the application models. It allows managing products and categories efficiently in the admin interface. This helps with user-friendliness.

```python
admin.py ×
products > admin.py > {} admin
1  from django.contrib import admin
2  #import model for products and categories
3  from .models import Product, Category
4
5  # class for extening django's built model admin feature
6  class ProductAdmin(admin.ModelAdmin):
7      # tuple showing which fields to display
8      list_display = (
9          'sku',
10         'name',
11         'category',
12         'rating',
13         'image',
14     )
15
16     ordering = ('sku',)
17
18
19 class CategoryAdmin(admin.ModelAdmin):
20     # tuple showing which fields to display
21     list_display = (
22         'friendly_name',
23         'name',
24     )
25 #registering the models for products and categories
26 admin.site.register(Product, ProductAdmin)
27 admin.site.register(Category, CategoryAdmin)
```

Imports:

- To customise the admin interface, it imports the Django admin module.
- The current app's Product and Category models are imported.

Classes:

- The Product model's specific ModelAdmin class is called ProductAdmin.
- The fields that will be shown in the Product list view are specified by list_display.
- ordering determines how the products are shown in the list view by default.
- The Category model has a special ModelAdmin class called CategoryAdmin.
- The fields that will be shown in the Category list view are specified by list_display.
- The relevant custom admin classes are registered for the Product and Category models.

Shopping bag:

Models:

No model was used.

URLS:

```
urls.py ×
shoppingbag > urls.py > {} admin
1  from django.contrib import admin
2  from django.urls import path
3  from . import views
4
5  urlpatterns = [
6      path('', views.view_bag, name='view_bag'),
7      path('add/<item_id>/', views.add_to_bag, name='add_to_bag'),
8      path('adjust/<item_id>/', views.adjust_bag, name='adjust_bag'),
9      path('remove/<item_id>/', views.remove_item, name='remove_item'),
10 ]
```

The admin site and other built-in Django applications are located in the django.contrib module.

django.urls:

- The Django URL manipulation tools are contained in this module.

from . import views:

- The views are imported from the current app using the from. import command. The view functions that process various HTTP requests and produce the necessary responses are contained in the views module.

path('', views.view_bag, name='view_bag'):

- The URL pattern for the app's shopping bag. Django calls the bag view function from the app's views.py module when a user accesses the root URL, and the view function processes the request and returns an HTTP response to render the shopping bag content.

path('add/<item_id>/', views.add_to_bag, name='add_to_bag'):

- This links URLs to the function add_to_bag. The add_to_bag function will receive the <item_id> as a URL argument.

path('adjust/<item_id>/', views.adjust_bag, name='adjust_bag'):

- This links URLs to the function adjust_bag. It makes use of the URL parameter item_id, just as above.

path('remove/<item_id>/', views.remove_item, name='remove_item'):

- This links the remove_item method to the URL. Once more, item_id is a URL parameter used.

Views:

Imports:

```
views.py ×
shoppingbag > views.py > render
1  from django.shortcuts import render, redirect, reverse, HttpResponse, get_object_or_404
2  from django.contrib import messages
3  from products.models import Product
4
5  # Create your views here
```

render:

- Render is a Django shorthand for generating template renderings and HTTP response backends.

redirect:

- The shortcuts module of Django offers the redirect feature. It's employed to carry out an HTTP redirect to a certain URL. This instructs the browser of the client to send a fresh request to the specified URL. It is frequently utilised in view functions to manage form submissions or to redirect users to a different page after a specific action.

Reverse:

- The shortcuts module of Django offers the reverse function. In the Django application, it is used to generate URLs for named URL patterns. The real URL is returned as a string after passing in the name of the URL pattern.

HttpResponse:

- The HTTP module of Django provides the class known as HTTPResponse. It symbolises an HTTP reply that may be returned to the client once a view is called.

get_object_or_404:

- The function get_object_or_404 retrieves an object from the database using its primary key (pk) and raises a 404 page not found error if the object is missing.

messages:

- A Django module is used to show users messages (such as success, error, and warning) while processing requests.

Product:

These models correspond to product database tables.

view_bag(request):

```
4
5   # Create your views here.
6   def view_bag(request):
7       """ A view to return the bag content page"""
8
9       return render(request, 'bag/bag.html')
0
1   # add in add to bag view
```

- The shopping bag content page ('bag/bag.html') is displayed using this view. It shows the items and their quantities that have been added to the shopping bag.

add_to_bag(request, item_id):

```
views.py ×
shoppingbag > views.py > add_to_bag
12  def add_to_bag(request, item_id):
13      """Add quanity of a product to the shopping bag"""
14      # add toast messages so strings will work
15      product = get_object_or_404(Product, pk=item_id)
16      # get quanity and convert it to an integar as it is a string
17      quantity = int(request.POST.get('quantity'))
18      redirect_url = request.POST.get('redirect_url')
19      # set size to none
20      size = None
21      #if statement for product size in request . post it gets set to that
22      if 'product_size' in request.POST:
23          size = request.POST['product_size']
24      # stores shopping bag in the sessions so it is not lost till the session is closed
25      bag = request.session.get('bag', {})
26
27      # if statment for sturcture of bag and check is size is been added
28      if size:
29          # dictionary for size using key work item by sizefor multiple sizes
30          if item_id in list(bag.keys()):
31              # check two items of same id and size exist and set it equal to quantity
32              if size in bag[item_id]['items_by_size'].keys():
33                  bag[item_id]['items_by_size'][size] += quantity
34                  # add string method for toast message
35                  messages.success(request, f"You have added size {size.upper()} {product.name} to your {bag[item_id]['items_by_size'][size]}!")
36                  bag[item_id]['items_by_size'][size] = quantity
37                  # add string method for toast message
38                  messages.success(request, f' You have added size {size.upper()} {product.name} to your shopping bag!')
39          else:
40              bag[item_id] = {'items_by_size': {size: quantity}}
41              # add string method for toast message
42              messages.success(request, f' You have added size {size.upper()} {product.name} to your shopping bag!')
43      #else if item has no size run orginal
44      else:
45          # add item to bag or update the bag quanity if already in bag
46          if item_id in list(bag.keys()):
47              bag[item_id] += quantity
48              # add string method for toast message
49              messages.success(request, f' You have updated {product.name} quantity to your {bag[item_id]}!')
50          else:
51              bag[item_id] = quantity
52              # add string method for toast message
53              messages.success(request, f' You have added {product.name} to your shopping bag!')
54
```

```
54
55      # overwrite the variable if it doesnt exisit
56      request.session['bag'] = bag
57      return redirect(redirect_url)
58
```

- It requires the two parameters item_id (the unique ID of the product to add) and request (the HTTP request object).
- The product with the specified item_id is retrieved from the database.
- It extracts the product's amount from the HTTP POST data and turns it into an integer.
- The page to which the user will be redirected after adding the item is obtained from the HTTP POST data as the redirect_url.Admin:
- If the product has a particular size, it determines whether that size is already present in the shopping bag. If so, the amount for that size is updated; if not, the product with the requested size and quantity is added.
- It checks to see if the item already exists in the bag and if the goods lack a specified size. If so, the quantity is updated; if not, the product is added to the stated quantity.
- It saves the updated bag in the user's session after adding or modifying the item in the bag.
- The user is then forwarded to the specified redirect_url.

adjust_bag(request, item_id):

```
views.py  ×
shoppingbag > views.py > adjust_bag
59   # view update product quantity in the shopping bag
60   def adjust_bag(request, item_id):
61       """ update quantity of a product to the shopping bag"""
62       # add toast messages so strings will work
63       product = get_object_or_404(Product, pk=item_id)
64       # get quanity and convert it to an integar as it is a string
65       quantity = int(request.POST.get('quantity'))
66       # set size to none
67       size = None
68       #if statement for product size in request . post it gets set to that
69       if 'product_size' in request.POST:
70           size = request.POST['product_size']
71       # stores shopping bag in the sessions so it is not lost till the session is closed
72       bag = request.session.get('bag', {})
73
74       # if statment for sturcture of bag and check is size is been added
75       if size:
76           # quantity is 0 gets updated accordingly
77           if quantity > 0:
78               # if there is a size it gets updated from the dictionary
79               bag[item_id]['items_by_size'][size] = quantity
80               # add string method for toast message
81               messages.success(request, f' You have updated size {size.upper()} {product.name} to your {bag[item_id]["items_by_size"][size]}!
82           else:
83               # del if quantity is 0
84               del bag[item_id]['items_by_size'][size]
85               if not bag[item_id]['item_by_size']:
86                   bag.pop(item_id)
87                   # add string method for toast message
88                   messages.success(request, f' You have removed size {size.upper()} {product.name} from your shopping bag!')
89       #else if item has no size run orginal
90       else:
91           # if no size and it get removed
92           if quantity > 0:
93               bag[item_id] = quantity
94               # add string method for toast message
95               messages.success(request, f' You have updated {product.name} quantity to your {bag[item_id]}!')
96           else:
97               bag.pop(item_id)
98               # add string method for toast message
99               messages.success(request, f' You have removed {product.name} from your shopping bag!')
100      # overwrite the variable if it doesnt exisit
101      request.session['bag'] = bag
```

- Request, an HTTP request object, and item_id, the specific ID of the product to be adjusted, are the two parameters it requires.
- The product with the specified item_id is retrieved from the database.
- It extracts the product's amount from the HTTP POST data and turns it into an integer.
- As a result, the product's size variable is set to None, indicating that it has no set size by default.
- If the product comes in a particular size, it determines whether that size is present in the POST data. If so, the size variable is updated with the supplied size.
- Request.session.get('bag',) is used to acquire the shopping bag dictionary from the user's session. It initialises an empty bag if there isn't one in the session.
- The function determines whether the quantity is larger than zero in the case where the product has a specific size. If the quantity is higher than 0, the quantity of that particular size in the bag is updated, and a success message is displayed. It removes that particular size from the bag if the amount is 0 or less, and the entire item from the bag if there are no more sizes available for that particular item.
- The function tests if the quantity is larger than 0 if the product has no specified size (size is None). If the quantity is more than 0, the quantity of that item in the bag is updated, and a success message is displayed. The entire item gets taken out of the bag if the amount is 0 or less.
- The revised bag is saved in the user's session after the bag has been adjusted.
- The user is then forwarded to the bag view page.

remove_item View:

```python
# view remove product quantity in the shopping bag
def remove_item(request, item_id):
    """ remove quantity of a product to the shopping bag"""
    # add toast messages so strings will work
    product = get_object_or_404(Product, pk=item_id)
    # add try block to return 200 error if code does not execure
    try:
        # set size to none, no need for quantity as it set to zero
        size = None
        #if statement for product size in request . post it gets set to that
        if 'product_size' in request.POST:
            size = request.POST['product_size']
        # stores shopping bag in the sessions so it is not lost till the session is closed
        bag = request.session.get('bag', {})

        # if statment for sturcture of bag and check is size is been added
        if size:
            # del if quantity is 0
            del bag[item_id]['items_by_size'][size]
            if not bag[item_id]['items_by_size']:
                bag.pop(item_id)
                # add string method for toast message
                messages.success(request, f' You have removed size {size.upper()} {product.name} from your shopping bag!')
        #else if item has no size run orginal
        else:
            # if no size and pop it out of the bag
                bag.pop(item_id)
                # add string method for toast message
                messages.success(request, f' You have removed {product.name} from your shopping bag!')
        # overwrite the variable if it doesnt exisit
        request.session['bag'] = bag
        return HttpResponse(status=200)
    #catch status 500 error. e will return error to template if anythign goes wrong
    except Exception as e:
        # error message
        messages.error(request, f'There has been an error removing this itme {e}')
        return HttpResponse(status=500)
```

- This view is used to remove items from the shopping bag.
- It only requires the HTTP request object named request as an argument.
- It tries to take the item from the bag that is indicated by the item_id and optional size.
- To show that things have been successfully removed, appropriate success messages are added.
- A 500 HTTP status response is provided if any errors are made while the process is running.

Admin:

No Admin file.

Contexts:

- The contexts.py file uses the Djangos context process and its named bag_contents for this application. A context processor in Django adds variables to the context of each template that is rendered. The context processor makes calculations and gives various information about the shopping bag here.

Imports:

```python
contexts.py ×
shoppingbag > contexts.py
1  from decimal import Decimal
2  from django.conf import settings
3  from django.shortcuts import get_object_or_404
4  from products.models import Product
```

Import Decimal:

- The Decimal class is imported on this line from the decimal module. To eliminate rounding mistakes in financial calculations, decimal is employed for accurate arithmetic operations using floating-point integers.

Import settings:

- Accessing Django project settings requires importing the settings module.

get_object_or_404:

- The function get_object_or_404 retrieves an object from the database using its primary key (pk) and raises a 404 page not found error if the object is missing.

Product:

- These models correspond to product database tables.

```
contexts.py ×
shoppingbag > contexts.py > Decimal
6   # makes all dictioanries avialble to all templeates on the app
7   def bag_contents(request):
8       # setting bag items to an empty list
9       bag_items = []
10      # total set to 0
11      total = 0
12      # count set to 0
13      product_count = 0
14      # accessing the bag in session if it exists
15      bag = request.session.get('bag', {})
16
17      # for loop for each item and quanity in the bag to add to bag
18      # add item_data to use all data not just quantity
19      for item_id, item_data in bag.items():
20          # if statment for code to be executed if it only has sizes.
21          # if item is an integer it knows to use quantity
22          if isinstance(item_data, int):
23              # getting the product
24              product = get_object_or_404(Product, pk=item_id)
25              # add price and quantity to total
26              total += item_data * product.price
27              product_count += item_data
28              # incremnts the product count by the quantity
29              bag_items.append({
30                  # dictonary added to bag
31                  'item_id': item_id,
32                  'quantity': item_data,
33                  'product': product,
34              })
35          # else it is a dictionary and it has to itterate accordingly through
36          else:
37              product = get_object_or_404(Product, pk=item_id)
38              for size, quantity in item_data['items_by_size'].items():
39                  total += quantity * product.price
40                  product_count += quantity
41                  bag_items.append({
42                      'item_id': item_id,
43                      'quantity': quantity,
44                      'product': product,
45                      'size': size,
46                  })
47
48      # calculation for free shipping and adding on cost of shipping
49      if total < settings.FREE_DELIVERY_THRESHOLD:
50          delivery = total * Decimal(settings.STANDARD_DELIVERY_PERCENTAGE/100)
```

Initializes variables:

bag_items:

- An empty list called bag_items will include dictionaries that describe the things in the shopping bag.

total:

- Starting with a value of 0, it will keep tabs on the total price of everything in the bag.

Product_count:

- Will keep track of the overall number of products in the bag and is initially set to 0.

bag:

- Retrieves the user's session's shopping bag. A dictionary with item IDs as keys and either quantities or dictionaries with sizes and quantities as values is anticipated to be included in the bag.

FOR Loop that loops through the items in the bag:

- If an item has a quantity (integer value), it advances the product count, adds the item to bag_items as a dictionary with the necessary information, and adds the price * quantity to the total.
- If an item has many sizes and quantities, the price and quantity for each size are added together to create the final price for the item. The product count is increased, and each item is added to bag_items as a dictionary containing size-specific information.

```
            })

    # calculation for free shipping and adding on cost of shipping
    if total < settings.FREE_DELIVERY_THRESHOLD:
        delivery = total * Decimal(settings.STANDARD_DELIVERY_PERCENTAGE/100)
        # shows users if they spend a little more they will be able to get free delivery
        free_delivery_delta = settings.FREE_DELIVERY_THRESHOLD - total
    else:
        delivery = 0
        free_delivery_delta = 0

    grand_total = delivery + total

    # returns a dictionary as context processer
    # makes context processor available in all templates
    context = {
        'bag_items': bag_items,
        'total': total,
        'product_count': product_count,
        'delivery': delivery,
        'free_delivery_delta': free_delivery_delta,
        'free_delivery_threshold': settings.FREE_DELIVERY_THRESHOLD,
        'grand_total': grand_total,

    }

    return context
```

Calculates shipping costs:

- Shipping costs are calculated by determining if the total worth of the items in the bag is less than the settings.FREE_DELIVERY_THRESHOLD. If yes, it determines the delivery fee as a % of the entire amount.
- Asa prompt to inform the user, it also determines the difference required to meet the free delivery threshold (free_delivery_delta).
- The delivery cost and the free_delivery_delta are both set to 0 if the overall value is equal to or more than the free delivery threshold.

Calculates the grand total:

- Get the final grand_total by adding the delivery fee to the total.

Returns a dictionary as the context:

- Various details about the shopping cart are included in the context dictionary, including bag_items, total, product_count, delivery, free_delivery_delta, free_delivery_threshold, and grand_total.

Checkout:

Models:

The models.py file located in the checkout application is for orders and order line item models. They together reflect the data structure used by an e-commerce website to manage orders and line items.

```
models.py ×
checkout > models.py > ...
1  # generates order numbers
2  import uuid
3  from django.db import models
4  from django.db.models import Sum
5  from django.conf import settings
6  from django_countries.fields import CountryField
7  from products.models import Product
8  from profiles.models import UserProfile
9
```

Imports:

import uuid:

- Importing the uuid module will enable you to create unique order numbers by using UUID.

models:

- Importing the model's module of Django to construct database models from django.db.

Sum:

- Importing the Sum aggregation method to compute the sums of query sets from django.db.models.

from the import settings in django.conf:

- Accessing Django project settings requires importing the settings module.

import from django_countries.fields CountryField:

- Adding the CountryField as a model field to manage country selection.

Products:

- Importing the Product model from the products app is what models import.

from profiles.models import UserProfile:

- importing the UserProfile model from the profiles application, from profiles.models.

Order model:

```
class Order(models.Model):
    order_number = models.CharField(max_length=32, null=False, editable=False)
    user_profile = models.ForeignKey(UserProfile, on_delete=models.SET_NULL,
                                     null=True, blank=True, related_name='orders')
    full_name = models.CharField(max_length=50, null=False, blank=False)
    email = models.EmailField(max_length=254, null=False, blank=False)
    phone_number = models.CharField(max_length=20, null=False, blank=False)
    country = CountryField(blank_label='Country *', null=False, blank=False)
    postcode = models.CharField(max_length=20, null=True, blank=True)
    town_or_city = models.CharField(max_length=40, null=False, blank=False)
    street_address1 = models.CharField(max_length=80, null=False, blank=False)
    street_address2 = models.CharField(max_length=80, null=True, blank=True)
    county = models.CharField(max_length=80, null=True, blank=True)
    date = models.DateTimeField(auto_now_add=True)
    delivery_cost = models.DecimalField(max_digits=6, decimal_places=2, null=False, default=0)
    order_total = models.DecimalField(max_digits=10, decimal_places=2, null=False, default=0)
    grand_total = models.DecimalField(max_digits=10, decimal_places=2, null=False, default=0)
    original_bag = models.TextField(null=False, blank=False, default='')
    stripe_pid = models.CharField(max_length=254, null=False, blank=False, default='')

    # returns the random order number
    def _generate_order_number(self):
        """
        Generate a random, unique order number using UUID
        """
        return uuid.uuid4().hex.upper()

    # function to update the total and update delivery cost
    def update_total(self):
        """
        Update grand total each time a line item is added,
        accounting for delivery costs.
        """
        # uses the aggregate function to update the line item by using sum method
        self.order_total = self.lineitems.aggregate(Sum('lineitem_total'))['lineitem_total__sum'] or 0
        # calculates the delivery cost
        if self.order_total < settings.FREE_DELIVERY_THRESHOLD:
            self.delivery_cost = self.order_total * settings.STANDARD_DELIVERY_PERCENTAGE / 100
        else:
            # set to 0 if higher than threshold
            self.delivery_cost = 0
        # gets the grand total
        self.grand_total = self.order_total + self.delivery_cost
        self.save()
```

```
        self.save()

    #   overwrites the default save method and returns random number fromm
    #   generate number function
    def save(self, *args, **kwargs):
        """
        Override the original save method to set the order number
        if it hasn't been set already.
        """
        if not self.order_number:
            self.order_number = self._generate_order_number()
        super().save(*args, **kwargs)

    # string to return order number
    def __str__(self):
        return self.order_number
```

- In the application, this object represents an order.
- Contains fields for order information including the order number, user profile, shipping details, date, delivery cost, order total, and overall total.
- Creates a unique order number at random using the uuid.
- Contains a special technique to modify the order total and delivery price based on the line item totals.
- If the order number hasn't been specified before storing the order, it will be set by overwriting the default save procedure.

OrderLineItem model:

```python
# class for indivdual line items, will relate shopping bag items to orders number
class OrderLineItem(models.Model):
    order = models.ForeignKey(Order, null=False, blank=False, on_delete=models.CASCADE, related_name='lineitems')
    product = models.ForeignKey(Product, null=False, blank=False, on_delete=models.CASCADE)
    product_size = models.CharField(max_length=2, null=True, blank=True) # XS, S, M, L, XL
    quantity = models.IntegerField(null=False, blank=False, default=0)
    lineitem_total = models.DecimalField(max_digits=6, decimal_places=2, null=False, blank=False, editable=False)

    def save(self, *args, **kwargs):
        """
        Override the original save method to set lineitem total
        and update order.
        """
        # multiplies the product by the price for each line item
        self.lineitem_total = self.product.price * self.quantity
        super().save(*args, **kwargs)

    # string to return sku and order number for each line item
    def __str__(self):
        return f'SKU: {self.product.sku} on order {self.order.order_number}'
```

- Represents a single order line that is connected to a product.
- Includes fields for the related order, item, size, quantity, and line item total.
- Computes the line item total using the product's price and quantity.
- The line item total is calculated and set when saving the line item, overriding the save function.

URLS:

Imports:

```python
urls.py
checkout > urls.py > {} path
1  from django.urls import path
2  from . import views
3  from .webhooks import webhook
4
```

django.urls:

- The Django URL manipulation tools are contained in this module.

from . import views:

- The views are imported from the current app using the from. import command. The view functions that process various HTTP requests and produce the necessary responses are contained in the views module.

From .webhooks:

- Imports the webhooks function form the webhook module.

```
urlpatterns = [
    # url to the check out funciton
    path('', views.checkout, name='checkout'),
    # checkout success url
    path('checkout_sucess/<order_number>', views.checkout_success, name='checkout_success'),
    # checkout data url
    path('cache_checkout_data/', views.cache_checkout_data, name='cache_checkout_data'),
    # webhook url
    path('wh/', webhook, name='webhook'),
]
```

Paths:

path('', views.checkout, name='checkout'):

- connects the root URL to the views module's checkout view method.
- A user will start the checkout procedure using this URL.
- path('checkout_sucess/<order_number>', views.checkout_success, name='checkout_success'):
- URLs that map, to the checkout_success view function, enter the order number.
- A URL parameter called "order_number" will be supplied to the checkout_success view method.
- After the checkout is complete, this URL will be used to display a success page that includes the order_number.

path('cache_checkout_data/', views.cache_checkout_data, name='cache_checkout_data'):

- Maps the cache_checkout_data view function to the URL.
- Before completing the payment, this URL will be used to cache the checkout information.

path('wh/', webhook, name='webhook'):

- Maps the URL from the webhooks module to the webhook function.
- The handling of incoming webhook events connected to payment processing or other external services will take place at this URL.

Views:

The checkout view handles the checkout process for the application, while integrating stripe payment API for secure payments and provides feedback to users. It will provide messages during checkout and order confirmation.

```python
from django.shortcuts import render, redirect, reverse, get_object_or_404, HttpResponse
from django.views.decorators.http import require_POST
from django.contrib import messages
from django.conf import settings

from .forms import OrderForm
from .models import Order, OrderLineItem

from products.models import Product
from profiles.models import UserProfile
from profiles.forms import UserProfileForm
from shoppingbag.contexts import bag_contents

import stripe
import json
```

Imports:

render:

- Render is a Django shorthand for generating template renderings and HTTP response backends.

redirect:

- The shortcuts module of Django offers the redirect feature. It's employed to carry out an HTTP redirect to a certain URL. This instructs the browser of the client to send a fresh request to the specified URL. It is frequently utilised in view functions to manage form submissions or to redirect users to a different page after a specific action.

Reverse:

- The shortcuts module of Django offers the reverse function. In the Django application, it is used to generate URLs for named URL patterns. The real URL is returned as a string after passing in the name of the URL pattern.

HttpResponse:

- The HTTP module of Django provides the class known as HTTPResponse. It symbolises an HTTP reply that may be returned to the client once a view is called.

get_object_or_404:

- The function get_object_or_404 retrieves an object from the database using its primary key (pk) and raises a 404 page not found error if the object is missing.

from django.views.decorators.http import require_POST

- Using the Django decorator require_POST, you can make sure that a view only permits HTTP POST requests and denies GET, PUT, and DELETE. It limits the HTTP methods that are permitted for a certain view to just POST.

messages:

- A Django module is used to show users messages (such as success, error, and warning) while processing requests.

Import settings:

- Accessing Django project settings requires importing the settings module.
- import OrderForm,Order, OrderLineItem

- Imports OrderForm, OrderLineItem, and other checkout-related forms and models.

Product:

- These models correspond to product database tables.

Imports user profile:

- it imports the models and associated forms from other the profile section of the application UserProfile.

from shoppingbag.contexts import bag_contents:

- Imports the bag_contents method from the shoppingbag package's contexts module.

Stripe Integration:

- To communicate with the Stripe API, the code imports the stripe module. Stripes payment processor is used to safely process payments made online.

cache_checkout_data:

```
# Create your views here.
# view for caching the meta data from billing and shipping info
@require_POST
# uses the POST method
def cache_checkout_data(request):
    try:
        pid = request.POST.get('client_secret').split('_secret')[0]
        stripe.api_key = settings.STRIPE_SECRET_KEY
        # calls stripe payment intent and what to modify
        stripe.PaymentIntent.modify(pid, metadata={
            # use json dump of shopping bag
            'bag': json.dumps(request.session.get('bag', {})),
            # if the info needs to be saved
            'save_info': request.POST.get('save_info'),
            # user who is placing order
            'username': request.user,
        })
        # return 200 response if runs okay
        return HttpResponse(status=200)
    # if there is an error return this message
    except Exception as e:
        messages.error(request, 'Sorry, your payment cannot be \
            processed right now. Please try again later.')
        return HttpResponse(content=e, status=400)

# checkout view
def checkout(request):
    # create stripe payment intent
    # public key
    stripe_public_key = settings.STRIPE_PUBLIC_KEY
    # secret key
    stripe_secret_key = settings.STRIPE_SECRET_KEY
    # check submit method is post and submit request
    if request.method == 'POST':
        bag = request.session.get('bag', {})

        # get form data from dictionary
        form_data = {
            'full_name': request.POST['full_name'],
            'email': request.POST['email'],
            'phone_number': request.POST['phone_number'],
            'country': request.POST['country'],
            'postcode': request.POST['postcode'],
            'town_or_city': request.POST['town_or_city'],
```

views.py ×

checkout > views.py > render

```python
            'postcode': request.POST['postcode'],
            'town_or_city': request.POST['town_or_city'],
            'street_address1': request.POST['street_address1'],
            'street_address2': request.POST['street_address2'],
            'county': request.POST['county'],
        }

        order_form = OrderForm(form_data)
        # if form is valid the order will get saved
        if order_form.is_valid():
            order = order_form.save(commit=False)
            pid = request.POST.get('client_secret').split('_secret')[0]
            order.stripe_pid = pid
            order.original_bag = json.dumps(bag)
            order.save()
            for item_id, item_data in bag.items():
                # try catch
                try:
                    # get product id out of the bag
                    product = Product.objects.get(id=item_id)
                    # if value is integer it dosent have a size
                    if isinstance(item_data, int):
                        order_line_item = OrderLineItem(
                            order=order,
                            product=product,
                            quantity=item_data,
                        )
                        order_line_item.save()
                    else:
                        # else has size will itterate through
                        for size, quantity in item_data['items_by_size'].items():
                            order_line_item = OrderLineItem(
                                order=order,
                                product=product,
                                quantity=quantity,
                                product_size=size,
                            )
                            order_line_item.save()
                # if product doesnt exist the error message will appear
                except Product.DoesNotExist:
                    messages.error(request, (
                        "One of the products in your bag wasn't found in our database. "
                        "Please call us for assistance!")
                    )
```

views.py

checkout > views.py > render

```
101                             "Please call us for assistance!")
102                         )
103                         order.delete()
104                         return redirect(reverse('view_bag'))
105                 # add file info to the session and redierect to success page
106                 request.session['save_info'] = 'save-info' in request.POST
107                 # redirect to checout success pages
108                 return redirect(reverse('checkout_success', args=[order.order_number]))
109             else:
110                 # if order form is invalid redirect and give error message
111                 messages.error(request, 'There was an error with your form. \
112                     Please double check your information.')
113             # else return get request
114         else:
115             # get the bag from the session
116             bag = request.session.get('bag', {})
117             # if statment for logic around if bag is empty
118             if not bag:
119                 messages.error(request, "Your bag is currently empty")
120                 return redirect(reverse('products'))
121 
122             # create new variable for stripe payments so the old one is not overridden
123             current_bag = bag_contents(request)
124             # getting the total key from the current bag
125             total = current_bag['grand_total']
126             # round the total for stripe
127             stripe_total = round(total * 100)
128             # setting secret key on stripe
129             stripe.api_key = stripe_secret_key
130             # createing the payment intent
131             intent = stripe.PaymentIntent.create(
132                 amount=stripe_total,
133                 currency=settings.STRIPE_CURRENCY,
134             )
135             # if the user is authenticated check
136             if request.user.is_authenticated:
137                 try:
138                     # if authenticated pre fill fields
139                     profile = UserProfile.objects.get(user=request.user)
140                     order_form = OrderForm(initial={
141                         'full_name': profile.user.get_full_name(),
142                         'email': profile.user.email,
143                         'phone_number': profile.default_phone_number,
144                         'country': profile.default_country,
```

Ln 1, Col 1

```
ws.py  ×
out > views.py > render
                'phone_number': profile.default_phone_number,
                'country': profile.default_country,
                'postcode': profile.default_postcode,
                'town_or_city': profile.default_town_or_city,
                'street_address1': profile.default_street_address1,
                'street_address2': profile.default_street_address2,
                'county': profile.default_county,
            })
        except UserProfile.DoesNotExist:
            order_form = OrderForm()
    else:
        order_form = OrderForm()
if not stripe_public_key:
    messages.warning(
        request, 'Stripe public key is unavailable, please set before continuing')
template = 'checkout/checkout.html'
context = {
    'order_form': order_form,
    # add stripe secret key
    'stripe_public_key': stripe_public_key,
    'client_secret': intent.client_secret,
}

# renders out the order form
return render(request, template, context)
```

- This view only allows POST queries, as indicated by the decoration @require_POST.
- It is in charge of caching metadata for the Stripe Payment Intent's checkout process.
- To change the Payment Intent on the Stripe platform, it takes the Stripe Payment Intent ID (client secret) from the POST request.
- As metadata for the Payment Intent, it saves the contents of the shopping bag, the user's decision to save information for future transactions, and the username of the person placing the order.
- If the update is effective, an HTTP 200 response is returned. Otherwise, it detects any process exceptions and sends back an HTTP 400 response along with an error message.

checkout:

- both GET and POST requests are handled.
- The user's information on the order form is verified if the request method is POST, and an order is then created in the database.
- After that, the view loops through the items in the shopping bag and generates OrderLineItem objects for each item in the order.
- The user is taken back to the shopping bag view and an error notice is displayed if the item in the bag does not exist in the database.
- The user is routed to the checkout_success view to display the order confirmation if the order is successfully processed, and the user's preference to save information for further orders is stored in the session.
- If the form is invalid, it displays an error message on the checkout page.

checkout_success:

```
views.py ×
checkout > views.py > render
169    # checkout success view to let user know order is complete
170    def checkout_success(request, order_number):
171        """
172        Handle successful checkouts
173        """
174        # check user wants ot save session
175        save_info = request.session.get('save_info')
176        # user oder number ot create order
177        order = get_object_or_404(Order, order_number=order_number)
178
179        if request.user.is_authenticated:
180            profile = UserProfile.objects.get(user=request.user)
181            # Attach the user's profile to the order
182            order.user_profile = profile
183            order.save()
184
185            # Save the user's info
186            if save_info:
187                profile_data = {
188                    'default_phone_number': order.phone_number,
189                    'default_country': order.country,
190                    'default_postcode': order.postcode,
191                    'default_town_or_city': order.town_or_city,
192                    'default_street_address1': order.street_address1,
193                    'default_street_address2': order.street_address2,
194                    'default_county': order.county,
195                }
196                user_profile_form = UserProfileForm(profile_data, instance=profile)
197                if user_profile_form.is_valid():
198                    user_profile_form.save()
199
200        # success message and order number noted
201        messages.success(request, f'Order successfully processed! \
202            Your order number is {order_number}. A confirmation \
203            email will be sent to {order.email}.')
204        # delete users bag from the session
205        if 'bag' in request.session:
206            del request.session['bag']
207        # setting the tempalte and context
208        template = 'checkout/checkout_success.html'
209        context = {
210            'order': order,
211        }
212        # return the template
213        return render(request, template, context)
```

- Following successful order placement, this view is displayed.
- Using the order_number provided in the URL (args=[order.order_number]), it obtains the order from the database.
- The user's profile is linked to the order if they are authenticated (logged in) and their shipping information is saved if they wish to do so for future orders.
- Along with the order number and an email confirmation, a success message that the order was handled successfully is displayed.
- To make room for the user's next purchase, the session's shopping bag gets cleaned.

Signals:

Functions known as signal handlers are called when specific events occur in the application. When an event occurs, they are utilised to carry out extra tasks or update relevant data. The application implements two signal handlers for the OrderLineItem model utilising the post_save and post_delete signals.

```python
from django.db.models.signals import post_save, post_delete
from django.dispatch import receiver
from .models import OrderLineItem

# exectues the update_on_save method
@receiver(post_save, sender=OrderLineItem)
# fucniton to handle signals from post save event
def update_on_save(sender, instance, created, **kwargs):
    """
    Update order total on lineitem update/create
    """
    instance.order.update_total()

# exectues the update_on_save method
@receiver(post_delete, sender=OrderLineItem)
# fucniton to handle signals from post save event
def update_on_delete(sender, instance, **kwargs):
    """
    Update order total on lineitem delete
    """
    instance.order.update_total()
```

Imports

from django.db.models.signals import post_save, post_delete:

- To handle post-save certain Django imports are used.

from django.dispatch import receiver:

- This import connects signal handlers to signal sender functions via the decorator function receiver.

from .models import OrderLineItem:

- It imports the OrderLineItem model from the same application. Individual line items in an order are typically represented by this model.

update_on_save:

- The OrderLineItem model's post_save signal is attached to the signal handler procedure called update_on_save.
- This function is linked to the OrderLineItem model's post_save signal by the @receiver(post_save, sender=OrderLineItem) decorator.
- Every time an existing OrderLineItem instance is modified or a new instance is generated like a save operation, the function is called.
- When this signal handler is invoked, it calls the corresponding Order instance's update_total() method to update the order total, which is the sum of the prices of all the line items in the order.

update_on_delete:

- The OrderLineItem model's post_delete signal is related to this signal handler function.

- This function is linked to the OrderLineItem model's post_delete signal via the @receiver(post_delete, sender=OrderLineItem) decorator.
- Every time an OrderLineItem instance is deleted, the function is invoked.
- When this signal handler is invoked, it runs the corresponding Order instance's update_total() method to update the order total following the deletion of the line item.

Webhooks:

The webhooks.py file handles incoming webhooks from Stripe.

```
webhooks.py ×
checkout > webhooks.py > webhook
1   from django.conf import settings
2   from django.http import HttpResponse
3   from django.views.decorators.http import require_POST
4   from django.views.decorators.csrf import csrf_exempt
5
6   from checkout.webhook_handler import StripeWH_Handler
7
8   import stripe
```

Import settings:

- Accessing Django project settings requires importing the settings module.

HttpResponse:

- The HTTP module of Django provides the class known as HTTPResponse. It symbolises an HTTP reply that may be returned to the client once a view is called.

from django.views.decorators.http import require_POST:

- Importing the require_POST decorator from django.views.decorators.http ensures that the view function can only be accessible through an HTTP POST request. It aids in preventing unauthorised access to the webhook.

from django.views.decorators.csrf import csrf_exemp

- Cross-Site Request Forgery (CSRF) protection is removed from the view function by using the csrf_exempt decorator, which is imported in this case. Because CSRF tokens are not normally included in webhooks, this decorator enables the view to get around CSRF security.

from checkout.webhook_handler import StripeWH_Handler:

- This import is used to bring in the StripeWH_Handler class from the webhook_handler.py module of the checkout app. The StripeWH_Handler class is a unique webhook handler that manages webhook events coming from the Stripe payment gateway.

Webhook:

checkout > webhooks.py > webhook

```python
@require_POST
@csrf_exempt
def webhook(request):
    """Listen for webhooks from Stripe"""
    # Setup
    wh_secret = settings.STRIPE_WH_SECRET
    stripe.api_key = settings.STRIPE_SECRET_KEY

    # Get the webhook data and verify its signature
    payload = request.body
    sig_header = request.META['HTTP_STRIPE_SIGNATURE']
    event = None

    try:
        event = stripe.Webhook.construct_event(
        payload, sig_header, wh_secret
        )
    except ValueError as e:
        # Invalid payload
        return HttpResponse(status=400)
    except stripe.error.SignatureVerificationError as e:
        # Invalid signature
        return HttpResponse(status=400)
    except Exception as e:
        return HttpResponse(content=e, status=400)

    # Set up a webhook handler
    handler = StripeWH_Handler(request)

    # Map webhook events to relevant handler functions
    event_map = {
        'payment_intent.succeeded': handler.handle_payment_intent_succeeded,
        'payment_intent.payment_failed': handler.handle_payment_intent_payment_failed,
    }

    # Get the webhook type from Stripe
    event_type = event['type']

    # If there's a handler for it, get it from the event map
    # Use the generic one by default
    event_handler = event_map.get(event_type, handler.handle_event)

    # Call the event handler with the event
    response = event_handler(event)
    return response
```

- Only POST requests are accepted by the view, and the CSRF protection is disregarded thanks to the @require_POST and @csrf_exempt decorators. Due to the lack of CSRF tokens in webhook requests, this is required for handling them.
- A Django request object that represents the incoming webhook request from Stripe is passed as a parameter to the method webhook(request).
- From the Django settings (settings.STRIPE_WH_SECRET and settings.STRIPE_SECRET_KEY), it extracts the Stripe webhook secret and API key.
- From the request, the payload and Stripe signature are taken. The HTTP_STRIPE_SIGNATURE header contains the Stripe signature.
- The code then uses the stripe.Webhook.construct_event() method to try and validate the signature. To retrieve the event data, this method verifies the signature and decodes the payload. If the verification is unsuccessful, several exceptions are triggered to address invalid payloads and signatures, returning a status code of 400 (Bad Request) in those circumstances.
- The code then creates a StripeWH_Handler object, which will handle the webhook events, if the signature can be correctly confirmed.
- Specific webhook events are mapped to matching handler functions in the StripeWH_Handler class using the event_map dictionary.

Webhooks Handler:

Imports:

```python
# webhook_handler.py
# checkout > webhook_handler.py > StripeWH_Handler
from django.http import HttpResponse
from django.core.mail import send_mail
from django.template.loader import render_to_string
from django.conf import settings

from .models import Order, OrderLineItem
from products.models import Product
from profiles.models import UserProfile

import json
import time

```

HttpResponse:

- The HTTP module of Django provides the class known as HTTPResponse. It symbolises an HTTP reply that may be returned to the client once a view is called.

django.core.mail.send_mail:

- Email notifications are sent using this feature. Users are frequently informed about their orders, progress updates, or any other pertinent information via this method.

django.template.loader.render_to_string:

- Using this function, a Django template is converted to a string. It is frequently employed to produce email content using HTML templates.

Import settings:

- Accessing Django project settings requires importing the settings module.

Imports user profile:

- it imports the models and associated forms from other the profile section of the application UserProfile.

Product:

- These models correspond to product database tables.

import OrderForm,Order, OrderLineItem

- Imports OrderForm, OrderLineItem, and other checkout-related forms and models.

JSON:

- The methods for working with JSON data are provided by this module. When communicating with external services or APIs, it's frequently used to parse incoming JSON data or serialise data into JSON format.

Time:

- This module offers some time-related features. When working with timestamps or adding delays to the code, for example, its use may vary.

StripeWH_Handler:

- Sends a confirmation email to the user with the order details using the function _send_confirmation_email.
- Handles a general, unforeseen, or unexpected webhook event. The webhook received message is included in the HTTP response that is returned.

handle_payment_intent_succeeded:

- Handles the payment_intent.succeeded webhook event from Stripe. It pulls information from the event data, makes any necessary adjustments to the user's profile, determines whether a previous order with identical details already exists in the database, and, if not, produces a new order. Additionally, the user receives a confirmation email. A message stating that the webhook was received and the result of the order creation is returned in an HTTP response.

handle_payment_intent_succeeded:

- Handles the payment_intent.payment_failed webhook event from Stripe using handle_payment_intent_payment_failed. The webhook received message is included in the HTTP response that is returned.

Admin:

For handling orders and order line items, use the admin configuration. In the Django admin interface, it enables admin users to see, add, and edit orders and the accompanying line items.

```
admin.py ×
checkout > admin.py > OrderAdmin > ordering
1   from django.contrib import admin
2   from .models import Order, OrderLineItem
3   # Register your models here.
4
5   # allows admin users to add and edit line items in admin
6   class OrderLineItemAdminInline(admin.TabularInline):
7       model = OrderLineItem
8       readonly_fields = ('lineitem_total',)
9
10  # class for order admin to generate administrion side
11  class OrderAdmin(admin.ModelAdmin):
12      inlines = (OrderLineItemAdminInline,)
13      # read only fields so they can not be update and change the order
14      fields = ('order_number', 'user_profile', 'date', 'full_name',
15                'email', 'phone_number', 'country',
16                'postcode', 'town_or_city', 'street_address1',
17                'street_address2', 'county', 'delivery_cost',
18                'order_total', 'grand_total', 'original_bag',
19                'stripe_pid')
20      # columns to show up in order list
21      list_display = ('order_number', 'date', 'full_name',
22                      'order_total', 'delivery_cost',
23                      'grand_total',)
24
25      # making sure orders will be ordered by date
26      ordering = ('-date',)
27
28  admin.site.register(Order, OrderAdmin)
```

Imports:

- To customise the admin interface, it imports the Django admin module.
- The current app's Order and OrderlineItem models are imported.

OrderLineItemAdminInline:

- The OrderLineItem model is represented by this inline admin class. It enables direct editing of order line items on the admin page for the order.
- The "lineitem_total" field is read-only because the readonly_fields attribute is set to ('lineitem_total'), which prevents users from directly updating it in the admin interface.

OrderAdmin:

- The Order model is represented by this admin class.
- For modifying order line items on the order admin page, the inlines property is set to (OrderLineItemAdminInline,), which also includes the inline class OrderLineItemAdminInline.
- The fields attribute specifies the fields that will be shown on the admin interface's order detail page. 'Order_number', 'user_profile', 'date', 'full_name', 'email', and other order-related information are included in these fields.
- The columns that will be shown in the admin interface's list view of orders are specified by the list_display property. 'Order_number', 'date', 'full_name', 'order_total', 'delivery_cost', and 'grand_total' are a few of the fields on it.
- The ordering element makes sure that the orders are displayed with the most recent orders listed first, in descending order according to the 'date' parameter.

admin.site.register(Order, OrderAdmin):

- The Order model and OrderAdmin class are registered with the admin site.

Profile:

Models:

When a new user is created or an existing user is saved, the Django model for user profiles (UserProfile) sets up a signal (post_save) to build or update user profiles.

```python
# models.py
# profiles > models.py > ...
from django.db import models
from django.contrib.auth.models import User
# imports neeeded for signals to work
from django.db.models.signals import post_save
from django.dispatch import receiver

from django_countries.fields import CountryField

# add user profile model to attach info for users profile
# user can only use one profile
class UserProfile(models.Model):
    """
    A user profile model for maintaining default
    delivery information and order history
    """
    # taken user models that are needed for profiles and add default before each
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    default_phone_number = models.CharField(max_length=20, null=True, blank=True)
    default_street_address1 = models.CharField(max_length=80, null=True, blank=True)
    default_street_address2 = models.CharField(max_length=80, null=True, blank=True)
    default_town_or_city = models.CharField(max_length=40, null=True, blank=True)
    default_county = models.CharField(max_length=80, null=True, blank=True)
    default_postcode = models.CharField(max_length=20, null=True, blank=True)
    default_country = CountryField(blank_label='Country', null=True, blank=True)
    # return username string
    def __str__(self):
        return self.user.username

# receiver to save user profiles details to file
# receiver to save user profiles details to file
@receiver(post_save, sender=User)
def create_or_update_user_profile(sender, instance, created, **kwargs):
    """
    Create or update the user profile
    """
    if created:
        UserProfile.objects.create(user=instance)
    # Existing users: just save the profile
    instance.userprofile.save()
```

Imports:

Models:

- Importing the model's module of Django to construct database models from django.db.

from profiles.models import User:

- importing the UserProfile model from the profiles application, from profiles.models.

from django.db.models.signals import post_save:

- To handle post-save certain Django imports are used.

from django.dispatch import receiver:

- This import connects signal handlers to signal sender functions via the decorator function receiver.

import from django_countries.fields CountryField:

- Adding the CountryField as a model field to manage country selection.

UserProfile Model:

- Additional user profile data, such as default shipping details and order history, are stored using this model.
- The user field uses a OneToOneField to establish a one-to-one link with the built-in Django User model. As a result, each user can only have a single user profile.

__str__(self) Method:

- The purpose of this method is to return a string that represents the UserProfile instance. It gives back the related user's username in this instance.

create_or_update_user_profile Signal Handler:

- The post_save signal of the User model is coupled to this method, which is a signal handler.
- This signal handler will be activated whenever a new User instance is created, and it will make a corresponding UserProfile object for that user.
- This signal handler will also be activated and save the related UserProfile if an existing person instance is saved (for example, when a person updates their profile).

URLS:

The URL for the views related to user profiles and order histories configures two URL patterns.

```
urls.py ×
profiles > urls.py > {} path
1  from django.urls import path
2  from . import views
3
4  urlpatterns = [
5      # url to link the proile view
6      path('', views.profile, name='profile'),
7      # view for order history
8      path('order_history/<order_number>', views.order_history, name='order_history'),
9  ]
```

Imports

django.urls:

The Django URL manipulation tools are contained in this module.

from . import views:

- The views are imported from the current app using the from. import command. The view functions that process various HTTP requests and produce the necessary responses are contained in the views module.

Path('', views.profile, name='profile'),

- The pattern 'profile' is mapped to the views module's profile view function.

path('order_history/<order_number>', views.order_history, name='order_history'),

- This Pattern 'order_history' is mapped to the views module's order_history view function. The view function can accept an argument in the form of the order_number> component.

Views:

The Django view for managing user profiles and order history.

Imports:

```
views.py ×
profiles > views.py > order_history
1    from django.shortcuts import render, get_object_or_404
2    from django.contrib import messages
3    from .models import UserProfile
4    from .forms import UserProfileForm
5    from checkout.models import Order
6    # Create your views here.
7    # create profile view returns html template
```

render:

- Render is a Django shorthand for generating template renderings and HTTP response backends.

get_object_or_404:

- The function get_object_or_404 retrieves an object from the database using its primary key (pk) and raises a 404 page not found error if the object is missing.

messages:

- A Django module is used to show users messages (such as success, error, and warning) while processing requests.

Imports user profile:

- it imports the models and associated forms from other the profile section of the application UserProfile.

Import Order:

- This enables the code to use the Order model in the current module and access the checkout properties and methods.

```python
# create profile view returns html template
def profile(request):
    """ Display the user's profile. """
    profile = get_object_or_404(UserProfile, user=request.user)

    if request.method == 'POST':
        form = UserProfileForm(request.POST, instance=profile)
        if form.is_valid():
            form.save()
            messages.success(request, 'Profile updated successfully')
    # populate user profile info
    form = UserProfileForm(instance=profile)
    # render order history
    orders = profile.orders.all()
    # link to profiles template url
    template = 'profiles/profile.html'
    context = {
        'form': form,
        'orders': orders,
        'on_profile_page': True
    }

    return render(request, template, context)
```

profile view:

- The user's profile information is displayed and updated via the profile view.
- It retrieves the UserProfile instance linked to the current user using the get_object_or_404 function. If the user profile is missing, a 404 Not Found answer will be given.
- The view determines whether the submitted form data is legitimate (form.is_valid()) when it receives a POST request and stores the changes if it is.
- It uses the Django messages framework to show a success message following a successful form submission.
- When rendering the profile template (profiles/profile.html), the view renders the UserProfileForm with the instance data filled in, the user's order history, and the flag on_profile_page set to True.

order_history View:

```
30
31  # view for order history
32  def order_history(request, order_number):
33      order = get_object_or_404(Order, order_number=order_number)
34      # message to show user is looking at a past order
35      messages.info(request, (
36          f'This is a past confirmation for order number {order_number}. '
37          'A confirmation email was sent on the order date.'
38      ))
39      # url to template checkout success
40      template = 'checkout/checkout_success.html'
41      context = {
42          'order': order,
43          'from_profile': True,
44      }
45
46      return render(request, template, context)
```

- The view displays the details of a past order based on the order_number passed as a URL parameter.
- The Order instance corresponding to the given order_number is retrieved using the get_object_or_404 function. It responds with a 404 not found error message if the order with the given number doesn't exist.
- To inform the user that they are seeing a previous order confirmation, the view displays an information message using Django's messages framework.
- It renders the Order object together with the flag from_profile set to True in the checkout/checkout_success.html template.

Admin:

No admin file.

2.3. Implementation

Models:

Models in Django serve as a representation of the application's data architecture. Classes like Product, Category, Order, UserProfiles, and checkout, which describe the database structure and the relationships between various entities, were frequently included in models in the application.

Homepage:

The homepage did not need a model as it was not adding anything to the database.

Products:

The products model showed how the database stored the categories and products and how they were structured. The Product model represents specific products with their features, such as name, description, and price, while the Category model is used to categorise products.

Category Model:

```python
# Create your models here.
# category model to seperate the products into categories
class Category(models.Model):
    # add meta class to change the spelling of categorys in admin to categories
    class Meta:
        verbose_name_plural = 'Categories'
    # name of the catgory to help make the name more code readable for views etc
    name = models.CharField(max_length=254)
    # friendly name of the category that will show up on the application
    friendly_name = models.CharField(max_length=254, null=True, blank=True)

    # string to take in the category name and return the name
    def __str__(self):
        return self.name

    # model method to returnt he friendly name if needed
    def get_friendly_name(self):
        return self.friendly_name
```

- The model's Meta class is used to alter the model's behaviour via the Django admin interface. In this instance, it transforms the model's plural name "Categories" into the correct "Categories."
- It serves as a representation of the product category and is utilised in the application to classify products.
- A character field (CharField) with a 254 character maximum is used for the name field. The name of the category is stored.
- A character field (CharField) with a 254 character limit is also used for the friendly_name field. It keeps a name for the category that may be seen in the user-friendly programme.
- __str__() method:
- The string representation of the model instance, which is the name of the category, is returned by the __str__() method.
- get_friendly_name() Method:
- A custom model method named get_friendly_name() can be used to access the friendly_name property of a Category instance. If the category's friendly_name has been set, it will be returned. It will return None if not.

Product Model:

```python
# product model
class Product(models.Model):
    # foreign key to call upon the category model, if category is deleted so if any models use it null will be set
    category = models.ForeignKey('Category', null=True, blank=True, on_delete=models.SET_NULL)
    sku = models.CharField(max_length=254, null=True, blank=True)
    name = models.CharField(max_length=254)
    description = models.TextField()
    # add sizes model so can selct sizes on certain products
    has_sizes = models.BooleanField(default=False, null=True, blank=True)
    price = models.DecimalField(max_digits=6, decimal_places=2)
    rating = models.DecimalField(max_digits=6, decimal_places=2, null=True, blank=True)
    image = models.ImageField(null=True, blank=True)

    # string method to return the product name same as category model
    def __str__(self):
        return self.name
```

- A product is represented by it in the application.
- A foreign key (ForeignKey) that connects the product to a category is the category field. It enables the product to be connected to a particular category, and if that category is

eliminated, the on_delete=models function is activated.Instead of removing the product, the SET_NULL option guarantees that the category will be set to null in the product.

- A character field (CharField) with a 254 character maximum is the SKU field. It is used as an identifier number for the products.json file in the fixtures folder.
- A character field (CharField) with a 254 character maximum is used for the name field. The product's name is kept there.
- A longer description of the product may be entered in the description box, which is a text field (TextField).
- Has_sizes is a boolean field (BooleanField) that specifies whether or not the product is available in various sizes. The product has sizes if True; else, it does not.
- A maximum of 6 digits and 2 decimal places can be stored in the price field (DecimalField), which is a decimal field.
- With a maximum of 6 digits and 2 decimal places, the rating field (DecimalField) stores the rating of the product. It is not necessary and can be null or blank.
- An image for the product may be uploaded using the image field. It is not necessary and can be null or blank.

__str__() method:

- The string representation of the model instance, which is the name of the category, is returned by the __str__() method.

Shopping Bag:

- The shopping bag used no model.

Checkout:

The Checkout model is for orders and order line item models. They together reflect the data structure used by an e-commerce website to manage orders and line items.

Order model:

```python
# Create your models here.
#  class to classify data need for each checkout
class Order(models.Model):
    order_number = models.CharField(max_length=32, null=False, editable=False)
    user_profile = models.ForeignKey(UserProfile, on_delete=models.SET_NULL,
                                     null=True, blank=True, related_name='orders')
    full_name = models.CharField(max_length=50, null=False, blank=False)
    email = models.EmailField(max_length=254, null=False, blank=False)
    phone_number = models.CharField(max_length=20, null=False, blank=False)
    country = CountryField(blank_label='Country *', null=False, blank=False)
    postcode = models.CharField(max_length=20, null=True, blank=True)
    town_or_city = models.CharField(max_length=40, null=False, blank=False)
    street_address1 = models.CharField(max_length=80, null=False, blank=False)
    street_address2 = models.CharField(max_length=80, null=True, blank=True)
    county = models.CharField(max_length=80, null=True, blank=True)
    date = models.DateTimeField(auto_now_add=True)
    delivery_cost = models.DecimalField(max_digits=6, decimal_places=2, null=False, default=0)
    order_total = models.DecimalField(max_digits=10, decimal_places=2, null=False, default=0)
    grand_total = models.DecimalField(max_digits=10, decimal_places=2, null=False, default=0)
    original_bag = models.TextField(null=False, blank=False, default='')
    stripe_pid = models.CharField(max_length=254, null=False, blank=False, default='')

    # returns the random order number
    def _generate_order_number(self):
        """
        Generate a random, unique order number using UUID
        """
        return uuid.uuid4().hex.upper()

    # function to update the total and update delivery cost
    def update_total(self):
        """
        Update grand total each time a line item is added,
        accounting for delivery costs.
        """
        # uses the aggregate function to update the line item by using sum method
        self.order_total = self.lineitems.aggregate(Sum('lineitem_total'))['lineitem_total__sum'] or 0
        # calculates the delivery cost
        if self.order_total < settings.FREE_DELIVERY_THRESHOLD:
            self.delivery_cost = self.order_total * settings.STANDARD_DELIVERY_PERCENTAGE / 100
        else:
            # set to 0 if higher than threshold
            self.delivery_cost = 0
```

models.py ✕

checkout > models.py > ...

```python
51              # set to 0 if higher than threshold
52              self.delivery_cost = 0
53          # gets the grand total
54          self.grand_total = self.order_total + self.delivery_cost
55          self.save()
56
57      #   overwrites the default save method and returns random number fromm
58      #   generate number function
59      def save(self, *args, **kwargs):
60          """
61          Override the original save method to set the order number
62          if it hasn't been set already.
63          """
64          if not self.order_number:
65              self.order_number = self._generate_order_number()
66          super().save(*args, **kwargs)
67
68      # string to return order number
69      def __str__(self):
70          return self.order_number
```

In

- Contains fields for order information including the order number, user profile, shipping details, date, delivery cost, order total, and overall total.
- Creates a unique order number at random using the uuid.
- Contains a special technique to modify the order total and delivery price based on the line item totals.

- If the order number hasn't been specified before storing the order, it will be set by overwriting the default save procedure.

OrderLineItem model:

```
# class for indivdual line items, will relate shopping bag items to orders number
class OrderLineItem(models.Model):
    order = models.ForeignKey(Order, null=False, blank=False, on_delete=models.CASCADE, related_name='lineitems')
    product = models.ForeignKey(Product, null=False, blank=False, on_delete=models.CASCADE)
    product_size = models.CharField(max_length=2, null=True, blank=True) # XS, S, M, L, XL
    quantity = models.IntegerField(null=False, blank=False, default=0)
    lineitem_total = models.DecimalField(max_digits=6, decimal_places=2, null=False, blank=False, editable=False)

    def save(self, *args, **kwargs):
        """
        Override the original save method to set lineitem total
        and update order.
        """
        # multiplies the product by the price for each line item
        self.lineitem_total = self.product.price * self.quantity
        super().save(*args, **kwargs)

    # string to return sku and order number for each line item
    def __str__(self):
        return f'SKU: {self.product.sku} on order {self.order.order_number}'
```

- Represents a single order line that is connected to a product.
- Includes fields for the related order, item, size, quantity, and line item total.
- Computes the line item total using the product's price and quantity.
- The line item total is calculated and set when saving the line item, overriding the save function.

Profile:

- The profile model is for creating new users or adding to an existing user's saved data. The Django model for user profiles (UserProfile) sets up a signal (post_save) to build or update user profiles.

UserProfile Model:

```
6
7   from django_countries.fields import CountryField
8
9   # add user profile model to attach info for users profile
10  # user can only use one profile
11  class UserProfile(models.Model):
12      """
13      A user profile model for maintaining default
14      delivery information and order history
15      """
16      # taken user models that are needed for profiles and add default before each
17      user = models.OneToOneField(User, on_delete=models.CASCADE)
18      default_phone_number = models.CharField(max_length=20, null=True, blank=True)
19      default_street_address1 = models.CharField(max_length=80, null=True, blank=True)
20      default_street_address2 = models.CharField(max_length=80, null=True, blank=True)
21      default_town_or_city = models.CharField(max_length=40, null=True, blank=True)
22      default_county = models.CharField(max_length=80, null=True, blank=True)
23      default_postcode = models.CharField(max_length=20, null=True, blank=True)
24      default_country = CountryField(blank_label='Country', null=True, blank=True)
25      # return username string
26      def __str__(self):
27          return self.user.username
28
```

- Additional user profile data, such as default shipping details and order history, are stored using this model.
- The user field uses a OneToOneField to establish a one-to-one link with the built-in Django User model. As a result, each user can only have a single user profile.
- __str__(self) Method:
- The purpose of this method is to return a string that represents the UserProfile instance. It gives back the related user's username in this instance.

create_or_update_user_profile Signal Handler:

```
28
29  # receiver to save user profiles details to file
30  # receiver to save user profiles details to file
31  @receiver(post_save, sender=User)
32  def create_or_update_user_profile(sender, instance, created, **kwargs):
33      """
34      Create or update the user profile
35      """
36      if created:
37          UserProfile.objects.create(user=instance)
38      # Existing users: just save the profile
39      instance.userprofile.save()
```

- The post_save signal of the User model is coupled to this method, which is a signal handler.
- This signal handler will be activated whenever a new User instance is created, and it will make a corresponding UserProfile object for that user.
- This signal handler will also be activated and save the related UserProfile if an existing person instance is saved (for example, when a person updates their profile).

Views:

The views handle all the logic of the application for processing incoming requests and returning an HTTP response.

The homepage view:

A web application's homepage requests are handled by the homepage view. When users visit the website, it is in charge of rendering the index.html template and displaying it as the web application's home page.

views.py

homepage > views.py > render

```python
from django.shortcuts import render

# Create your views here.

def index(request):
    """ A view to return the index.html page"""

    return render(request, 'home/index.html')
```

def **index(request):**

- This is a view function called index. It is in charge of responding to HTTP requests from the homepage URL.

return render(request, 'home/index.html'):

returns:

- This line renders the index.html template and returns it as an HTTP response using the render function.

request:

- The user's HTTP request object includes details about the browser they used, any session data, and any supplied data (GET/POST parameters).

'home/index.html':

- This is the relative path to the index file within the templates directory of the homepage app. It instructs Django to use the templates folder of the app's home directory's index.html template.

Products view:

The product views functions manage and show products and the product details within the application.While the 'product_detail.html' template is responsible for displaying specific product details, the 'products.html' template is responsible for displaying a list of products.

products_view(request):

```python
def products_view(request):
    """ A view to show all products, including sorting and search queries """

    # return products form databse
    products = Product.objects.all()
    #stops an error wehn loading products page without a search term
    query = None
    #query for returning selected products by categories
    categories = None
    # query for returning sproducts in sort query
    sort = None
    # query for returning sproducts in direction query
    direction = None

    # checking if requests. GET exists
    if request.GET:

        # if statment to sort by pricing, rating or category in a certain direction
        if 'sort' in request.GET:
            sortkey = request.GET['sort']
            sort = sortkey
            # annotate to allow case sensitive sorting
            if sortkey == 'name':
                sortkey = 'lower_name'
                products = products.annotate(lower_name=Lower('name'))

            # condtional to see if sort key is equal to category and if is return name
            if sortkey == 'category':
                sortkey = 'category__name'

            if 'direction' in request.GET:
                direction = request.GET['direction']
                if direction == 'desc':
                    sortkey = f'-{sortkey}'
            products = products.order_by(sortkey)

        # if statment for selected categories to be returned
        if 'category' in request.GET:
            categories = request.GET['category'].split(',')
            #filter products down to only the categories that are in the list
            products = products.filter(category__name__in=categories)
            # filter all categories
            categories = Category.objects.filter(name__in=categories)

```

- This view function manages the products page, which showcases all of the products. It has features for categorising and sorting products according to search terms and chosen categories.
- The code initially uses the database to retrieve all products using Product.objects.all()
- It determines whether any query parameters are present in the GET request and then takes the appropriate steps following those parameters. For instance, it may filter products based on certain categories and search terms, and it can sort products by name, category, cost, or rating.
- The products variable holds the sorted and filtered products.
- The function additionally receives and processes information from the GET request regarding search queries (q), selected categories (category), sorting, and direction (sort and direction).
- The filtered products, search terms, chosen categories, and current sorting options are used to generate the context dictionary.
- The 'products.html' template is then rendered with the context and returned as the HTTP response by the function.

product_detail(request, product_id):

```
79
80  def product_detail(request, product_id):
81      """ A view to single product details """
82
83      # taking in project id and getting object 404
84      product = get_object_or_404(Product, pk=product_id)
85
86      context = {
87          'product': product,
88      }
89      # return on product details
90      return render(request, 'products/product_detail.html', context)
```

- The product detail page, which displays comprehensive details about a single product, is handled by this view function.
- The get_object_or_404 function is used by the function to retrieve the specific product with the specified ID from the database. The function accepts the argument product_id.
- If the specified product ID is invalid, a 404 page not found message is returned.
- The product variable holds the product object.
- The product information is used to generate the context dictionary.
- The 'product_detail.html' template is then rendered with the context and returned as the HTTP response by the function.

Shopping bag view:

The shopping bag view manages all shopping bag functionality in the application. It allows users to add, update, and remove items form there shopping bags as required.

view_bag(request):

```
# Create your views here.
def view_bag(request):
    """ A view to return the bag content page"""

    return render(request, 'bag/bag.html')
```

- The shopping bag content page ('bag/bag.html') is displayed using this view. It shows the items and their quantities that have been added to the shopping bag.

add_to_bag(request, item_id):

```python
views.py ×
shoppingbag > views.py > add_to_bag
12  def add_to_bag(request, item_id):
13      """Add quanity of a product to the shopping bag"""
14      # add toast messages so strings will work
15      product = get_object_or_404(Product, pk=item_id)
16      # get quanity and convert it to an integar as it is a string
17      quantity = int(request.POST.get('quantity'))
18      redirect_url = request.POST.get('redirect_url')
19      # set size to none
20      size = None
21      #if statement for product size in request . post it gets set to that
22      if 'product_size' in request.POST:
23          size = request.POST['product_size']
24      # stores shopping bag in the sessions so it is not lost till the session is closed
25      bag = request.session.get('bag', {})
26
27      # if statment for sturcture of bag and check is size is been added
28      if size:
29          # dictionary for size using key work item by sizefor multiple sizes
30          if item_id in list(bag.keys()):
31              # check two items of same id and size exist and set it equal to quantity
32              if size in bag[item_id]['items_by_size'].keys():
33                  bag[item_id]['items_by_size'][size] += quantity
34                  # add string method for toast message
35                  messages.success(request, f"You have added size {size.upper()} {product.name} to your {bag[item_id]['items_by_size'][size]}!")
36                  bag[item_id]['items_by_size'][size] = quantity
37                  # add string method for toast message
38                  messages.success(request, f' You have added size {size.upper()} {product.name} to your shopping bag!')
39          else:
40              bag[item_id] = {'items_by_size': {size: quantity}}
41              # add string method for toast message
42              messages.success(request, f' You have added size {size.upper()} {product.name} to your shopping bag!')
43      #else if item has no size run orginal
44      else:
45          # add item to bag or update the bag quanity if already in bag
46          if item_id in list(bag.keys()):
47              bag[item_id] += quantity
48              # add string method for toast message
49              messages.success(request, f' You have updated {product.name} quantity to your {bag[item_id]}!')
50          else:
51              bag[item_id] = quantity
52              # add string method for toast message
53              messages.success(request, f' You have added {product.name} to your shopping bag!')
54
```

- It requires the two parameters item_id (the unique ID of the product to add) and request (the HTTP request object).
- The product with the specified item_id is retrieved from the database.
- It extracts the product's amount from the HTTP POST data and turns it into an integer.
- The page to which the user will be redirected after adding the item is obtained from the HTTP POST data as the redirect_url.Admin:
- If the product has a particular size, it determines whether that size is already present in the shopping bag. If so, the amount for that size is updated; if not, the product with the requested size and quantity is added.
- It checks to see if the item already exists in the bag and if the goods lack a specified size. If so, the quantity is updated; if not, the product is added to the stated quantity.
- It saves the updated bag in the user's session after adding or modifying the item in the bag.
- The user is then forwarded to the specified redirect_url.

adjust_bag(request, item_id):

```
views.py ×
shoppingbag > views.py > add_to_bag
59  # view update product quantity in the shopping bag
60  def adjust_bag(request, item_id):
61      """ update quantity of a product to the shopping bag"""
62      # add toast messages so strings will work
63      product = get_object_or_404(Product, pk=item_id)
64      # get quanity and convert it to an integar as it is a string
65      quantity = int(request.POST.get('quantity'))
66      # set size to none
67      size = None
68      #if statement for product size in request . post it gets set to that
69      if 'product_size' in request.POST:
70          size = request.POST['product_size']
71      # stores shopping bag in the sessions so it is not lost till the session is closed
72      bag = request.session.get('bag', {})
73
74      # if statment for sturcture of bag and check is size is been added
75      if size:
76          # quantity is 0 gets updated accordingly
77          if quantity > 0:
78              # if there is a size it gets updated from the dictionary
79              bag[item_id]['items_by_size'][size] = quantity
80              # add string method for toast message
81              messages.success(request, f' You have updated size {size.upper()} {product.name} to your {bag[item_id]["items_by_size"][size]}!')
82          else:
83              # del if quantity is 0
84              del bag[item_id]['items_by_size'][size]
85              if not bag[item_id]['item_by_size']:
86                  bag.pop(item_id)
87                  # add string method for toast message
88                  messages.success(request, f' You have removed size {size.upper()} {product.name} from your shopping bag!')
89      #else if item has no size run orginal
90      else:
91          # if no size and it get removed
92          if quantity > 0:
93              bag[item_id] = quantity
94              # add string method for toast message
95              messages.success(request, f' You have updated {product.name} quantity to your {bag[item_id]}!')
96          else:
97              bag.pop(item_id)
98              # add string method for toast message
99              messages.success(request, f' You have removed {product.name} from your shopping bag!')
100     # overwrite the variable if it doesnt exisit
101     request.session['bag'] = bag
```

- Request, an HTTP request object, and item_id, the specific ID of the product to be adjusted, are the two parameters it requires.
- The product with the specified item_id is retrieved from the database.
- It extracts the product's amount from the HTTP POST data and turns it into an integer.
- As a result, the product's size variable is set to None, indicating that it has no set size by default.
- If the product comes in a particular size, it determines whether that size is present in the POST data. If so, the size variable is updated with the supplied size.
- Request.session.get('bag',) is used to acquire the shopping bag dictionary from the user's session. It initialises an empty bag if there isn't one in the session.
- The function determines whether the quantity is larger than zero in the case where the product has a specific size. If the quantity is higher than 0, the quantity of that particular size in the bag is updated, and a success message is displayed. It removes that particular size from the bag if the amount is 0 or less, and the entire item from the bag if there are no more sizes available for that particular item.
- The function tests if the quantity is larger than 0 if the product has no specified size (size is None). If the quantity is more than 0, the quantity of that item in the bag is updated, and a success message is displayed. The entire item gets taken out of the bag if the amount is 0 or less.
- The revised bag is saved in the user's session after the bag has been adjusted.
- The user is then forwarded to the bag view page.

remove_item View:

```
# view remove product quantity in the shopping bag
def remove_item(request, item_id):
    """ remove quantity of a product to the shopping bag"""
    # add toast messages so strings will work
    product = get_object_or_404(Product, pk=item_id)
    # add try block to return 200 error if code does not execure
    try:
        # set size to none, no need for quantity as it set to zero
        size = None
        #if statement for product size in request . post it gets set to that
        if 'product_size' in request.POST:
            size = request.POST['product_size']
        # stores shopping bag in the sessions so it is not lost till the session is closed
        bag = request.session.get('bag', {})

        # if statment for sturcture of bag and check is size is been added
        if size:
            # del if quantity is 0
            del bag[item_id]['items_by_size'][size]
            if not bag[item_id]['items_by_size']:
                bag.pop(item_id)
                # add string method for toast message
                messages.success(request, f' You have removed size {size.upper()} {product.name} from your shopping bag!')
        #else if item has no size run orginal
        else:
            # if no size and pop it out of the bag
                bag.pop(item_id)
                # add string method for toast message
                messages.success(request, f' You have removed {product.name} from your shopping bag!')
        # overwrite the variable if it doesnt exisit
        request.session['bag'] = bag
        return HttpResponse(status=200)
    #catch status 500 error. e will return error to template if anythign goes wrong
    except Exception as e:
        # error message
        messages.error(request, f'There has been an error removing this itme {e}')
        return HttpResponse(status=500)
```

- This view is used to remove items from the shopping bag.
- It only requires the HTTP request object named request as an argument.
- It tries to take the item from the bag that is indicated by the item_id and optional size.
- To show that things have been successfully removed, appropriate success messages are added.
- A 500 HTTP status response is provided if any errors are made while the process is running.

Checkout View:

The checkout page view manages the application's checkout process, integrates the Stripe payment API for safe payments, and gives users feedback. Messages will be displayed upon checkout and during order confirmation.

cache_checkout_data:

```
16
17  # Create your views here.
18  # view for caching the meta data from billing and shipping info
19  @require_POST
20  # uses the POST method
21  def cache_checkout_data(request):
22      try:
23          pid = request.POST.get('client_secret').split('_secret')[0]
24          stripe.api_key = settings.STRIPE_SECRET_KEY
25          # calls stripe payment intent and what to modify
26          stripe.PaymentIntent.modify(pid, metadata={
27              # use json dump of shopping bag
28              'bag': json.dumps(request.session.get('bag', {})),
29              # if the info needs to be saved
30              'save_info': request.POST.get('save_info'),
31              # user who is placing order
32              'username': request.user,
33          })
34          # return 200 response if runs okay
35          return HttpResponse(status=200)
36      # if there is an error return this message
37      except Exception as e:
38          messages.error(request, 'Sorry, your payment cannot be \
39              processed right now. Please try again later.')
40          return HttpResponse(content=e, status=400)
41
```

- This view only allows POST queries, as indicated by the decoration @require_POST.
- It is in charge of caching metadata for the Stripe Payment Intent's checkout process.
- To change the Payment Intent on the Stripe platform, it takes the Stripe Payment Intent ID (client secret) from the POST request.
- As metadata for the Payment Intent, it saves the contents of the shopping bag, the user's decision to save information for future transactions, and the username of the person placing the order.
- If the update is effective, an HTTP 200 response is returned. Otherwise, it detects any process exceptions and sends back an HTTP 400 response along with an error message.

checkout:

```python

# checkout view
def checkout(request):
    # create stripe payment intent
    # public key
    stripe_public_key = settings.STRIPE_PUBLIC_KEY
    # secret key
    stripe_secret_key = settings.STRIPE_SECRET_KEY
    # check submit method is post and submit request
    if request.method == 'POST':
        bag = request.session.get('bag', {})

        # get form data from dictionary
        form_data = {
            'full_name': request.POST['full_name'],
            'email': request.POST['email'],
            'phone_number': request.POST['phone_number'],
            'country': request.POST['country'],
            'postcode': request.POST['postcode'],
            'town_or_city': request.POST['town_or_city'],
            'street_address1': request.POST['street_address1'],
            'street_address2': request.POST['street_address2'],
            'county': request.POST['county'],
        }

        order_form = OrderForm(form_data)
        # if form is valid the order will get saved
        if order_form.is_valid():
            order = order_form.save(commit=False)
            pid = request.POST.get('client_secret').split('_secret')[0]
            order.stripe_pid = pid
            order.original_bag = json.dumps(bag)
            order.save()
            for item_id, item_data in bag.items():
                # try catch
                try:
                    # get product id out of the bag
                    product = Product.objects.get(id=item_id)
                    # if value is integer it dosent have a size
                    if isinstance(item_data, int):
                        order_line_item = OrderLineItem(
```

- both GET and POST requests are handled.
- The user's information on the order form is verified if the request method is POST, and an order is then created in the database.
- After that, the view loops through the items in the shopping bag and generates OrderLineItem objects for each item in the order.
- The user is taken back to the shopping bag view and an error notice is displayed if the item in the bag does not exist in the database.
- The user is routed to the checkout_success view to display the order confirmation if the order is successfully processed, and the user's preference to save information for further orders is stored in the session.
- If the form is invalid, it displays an error message on the checkout page.

checkout_success:

```python
# checkout success view to let user know order is complete
def checkout_success(request, order_number):
    """
    Handle successful checkouts
    """
    # check user wants ot save session
    save_info = request.session.get('save_info')
    # user oder number ot create order
    order = get_object_or_404(Order, order_number=order_number)

    if request.user.is_authenticated:
        profile = UserProfile.objects.get(user=request.user)
        # Attach the user's profile to the order
        order.user_profile = profile
        order.save()

        # Save the user's info
        if save_info:
            profile_data = {
                'default_phone_number': order.phone_number,
                'default_country': order.country,
                'default_postcode': order.postcode,
                'default_town_or_city': order.town_or_city,
                'default_street_address1': order.street_address1,
                'default_street_address2': order.street_address2,
                'default_county': order.county,
            }
            user_profile_form = UserProfileForm(profile_data, instance=profile)
            if user_profile_form.is_valid():
                user_profile_form.save()

    # success message and order number noted
    messages.success(request, f'Order successfully processed! \
        Your order number is {order_number}. A confirmation \
        email will be sent to {order.email}.')
    # delete users bag from the session
    if 'bag' in request.session:
        del request.session['bag']
    # setting the tempalte and context
    template = 'checkout/checkout_success.html'
    context = {
        'order': order,
    }
    # return the template
    return render(request, template, context)
```

- Following successful order placement, this view is displayed.
- Using the order_number provided in the URL (args=[order.order_number]), it obtains the order from the database.
- The user's profile is linked to the order if they are authenticated (logged in) and their shipping information is saved if they wish to do so for future orders.
- Along with the order number and an email confirmation, a success message that the order was handled successfully is displayed.
- To make room for the user's next purchase, the session's shopping bag gets cleaned.

Signals:

- Functions known as signal handlers are called when specific events occur in the application. When an event occurs, they are utilised to carry out extra tasks or update relevant data. The application implements two signal handlers for the OrderLineItem model utilising the post_save and post_delete signals.

Profiles view:

The profiles view manages, the profiles view controls, the user's profile and displays their order history. The order_history view displays information about a specific past order when a user visits it from their profile, whereas the profile view enables users to read and change their profile information.

profile view:

```
# create profile view returns html template
def profile(request):
    """ Display the user's profile. """
    profile = get_object_or_404(UserProfile, user=request.user)

    if request.method == 'POST':
        form = UserProfileForm(request.POST, instance=profile)
        if form.is_valid():
            form.save()
            messages.success(request, 'Profile updated successfully')
    # populate user profile info
    form = UserProfileForm(instance=profile)
    # render order history
    orders = profile.orders.all()
    # link to profiles template url
    template = 'profiles/profile.html'
    context = {
        'form': form,
        'orders': orders,
        'on_profile_page': True
    }

    return render(request, template, context)
```

- The user's profile information is displayed and updated via the profile view.
- It retrieves the UserProfile instance linked to the current user using the get_object_or_404 function. If the user profile is missing, a 404 Not Found answer will be given.
- The view determines whether the submitted form data is legitimate (form.is_valid()) when it receives a POST request and stores the changes if it is.
- It uses the Django messages framework to show a success message following a successful form submission.
- When rendering the profile template (profiles/profile.html), the view renders the UserProfileForm with the instance data filled in, the user's order history, and the flag on_profile_page set to True.

order_history View:

```python
# view for order history
def order_history(request, order_number):
    order = get_object_or_404(Order, order_number=order_number)
    # message to show user is looking at a past order
    messages.info(request, (
        f'This is a past confirmation for order number {order_number}. '
        'A confirmation email was sent on the order date.'
    ))
    # url to template checkout success
    template = 'checkout/checkout_success.html'
    context = {
        'order': order,
        'from_profile': True,
    }

    return render(request, template, context)
```

- The view displays the details of a past order based on the order_number passed as a URL parameter.
- The Order instance corresponding to the given order_number is retrieved using the get_object_or_404 function. It responds with a 404 not found error message if the order with the given number doesn't exist.
- To inform the user that they are seeing a previous order confirmation, the view displays an information message using Django's messages framework.
- It renders the Order object together with the flag from_profile set to True in the checkout/checkout_success.html template.

Forms:

The homepage has no form.

Products have no form.

The shopping bag has no form.

Checkout:

The checkout form called OrderForm is associated with the order model. Which allows users to submit information for an order.

```python
# class for the order form
class OrderForm(forms.ModelForm):
    class Meta:
        # tells danjo which model is assoicated ot the form
        model = Order
        fields = ('full_name', 'email', 'phone_number',
                  'street_address1', 'street_address2',
                  'town_or_city', 'postcode', 'country',
                  'county',)

    # init method
    def __init__(self, *args, **kwargs):
        """
        Add placeholders and classes, remove auto-generated
        labels and set autofocus on first field
        """
        # dictionary of place holders so no empty boxes
        super().__init__(*args, **kwargs)
        placeholders = {
            'full_name': 'Full Name',
            'email': 'Email Address',
            'phone_number': 'Phone Number',
            'postcode': 'Postal Code',
            'town_or_city': 'Town or City',
            'street_address1': 'Street Address 1',
            'street_address2': 'Street Address 2',
            'county': 'County, State or Locality',
        }

        # add stars to to fields that are essential
        self.fields['full_name'].widget.attrs['autofocus'] = True
        for field in self.fields:
            if field != 'country':
                if self.fields[field].required:
                    placeholder = f'{placeholders[field]} *'
                else:
                    placeholder = placeholders[field]
                self.fields[field].widget.attrs['placeholder'] = placeholder
            self.fields[field].widget.attrs['class'] = 'stripe-style-input'
            self.fields[field].label = False
```

OrderForm:

- The class is defined using the form class.ModelForm. The form will be generated automatically using the Order model.
- The model is related to the fields to be included in the form and is specified by the Meta inner class within OrderForm.
- The __init__ method is overridden to alter the initialization behaviour of the form.
- To improve the user experience, form fields have placeholders and classes inserted inside the __init__ method. Additionally, the fields' auto-generated labels have been eliminated.
- The form fields' placeholders are stored in the placeholders dictionary. To help users decide what data to enter, these placeholders will be visible in the form inputs.
- The base class ModelForm is initialised with the supplied parameters by calling super().__init__(*args, **kwargs).
- Widgets are used for the form fields by using the self.fields[field].widget.attrs portion. Since the 'class' attribute in this instance is set to'stripe-style-input', this CSS class will be applied to each type of field.

- The field in self.fields. fields become buried when label = False is used to remove their automatically produced labels.

Profiles form:

The profiles form uses the UserProfile model, called UserProfileForm. Users can submit data for their user profiles.

```
forms.py ×
profiles > forms.py > ...
1   from django import forms
2   from .models import UserProfile
3
4   # class for the profiles form
5   class UserProfileForm(forms.ModelForm):
6       class Meta:
7           # tells danjo which model is assoicated ot the form
8           model = UserProfile
9           exclude = ('user',)
10
11      # init method
12      def __init__(self, *args, **kwargs):
13          """
14          Add placeholders and classes, remove auto-generated
15          labels and set autofocus on first field
16          """
17          # dictionary of place holders so no empty boxes
18          super().__init__(*args, **kwargs)
19          placeholders = {
20              'default_phone_number': 'Phone Number',
21              'default_postcode': 'Postal Code',
22              'default_town_or_city': 'Town or City',
23              'default_street_address1': 'Street Address 1',
24              'default_street_address2': 'Street Address 2',
25              'default_county': 'County, State or Locality',
26          }
27
28          # add stars to to fields that are essential
29          self.fields['default_phone_number'].widget.attrs['autofocus'] = True
30          for field in self.fields:
31              if field != 'default_country':
32                  if self.fields[field].required:
33                      placeholder = f'{placeholders[field]} *'
34                  else:
35                      placeholder = placeholders[field]
36                  self.fields[field].widget.attrs['placeholder'] = placeholder
37              self.fields[field].widget.attrs['class'] = 'border-black rounded-0 profile-form-input'
38              self.fields[field].label = False
```

UserProfileForm:

- The class is defined using the form class.ModelForm. The form will be generated automatically using the UserProfile model.
- The UserProfileForm's Meta inner class identifies the model it belongs to and the form fields that should be present. The 'user' field is excluded from the form using the exclude property. As a result, the 'user' field won't be included in the form and will instead be handled separately.
- The __init__ method is overridden to change the initialising behaviour of the form.
- To improve the user experience, form fields have placeholders and classes inserted inside the __init__ method.
- The form fields' placeholders are stored in the placeholders dictionary. To help users decide what data to enter, these placeholders are visible in the form inputs.
- The base class ModelForm is initialised with the supplied parameters by calling super().__init__(*args, **kwargs).

- Widgets are used for the form fields by using the self.fields[field].widget.attrs portion. Since the 'class' attribute in this instance is set to'stripe-style-input', this CSS class will be applied to each type of field.
- The field in self.fields. fields become buried when label = False is used to remove their automatically produced labels.

Sorting:

Code to sort products by price, category, or rating.

```
<a class="logo-font font-weight-bold nav-link text-black mr-5" href="#" id="all-products-link" data-toggle="dropdown" aria-haspopup="tru
    All Products
</a>
<div class="dropdown-menu border-0" aria-labelledby="all-products-link">
    <!-- sort products by price in accending order -->
    <a href="{% url 'products' %}?sort=price&direction=asc" class="dropdown-item">By Price</a>
    <!-- sort products by rating in desending order -->
    <a href="{% url 'products' %}?sort=rating&direction=desc" class="dropdown-item ">By Rating</a>
    <!-- sort products by vategory in accending order -->
    <a href="{% url 'products' %}?sort=category&direction=asc" class="dropdown-item ">By Category</a>
    <!-- sort productsby all products -->
    <a href="{% url 'products' %}" class="dropdown-item">All Products</a>
</div>
</li>
```

- “By Price" link: Sorts products by price in ascending order when clicked.
- "By Rating" link: Sorts products by rating in descending order when clicked.
- "By Category" link: Sorts products by category in ascending order when clicked.
- "All Products" link: Displays all products when clicked.

products_view:

- The products_view lists items with sorting and search features that will be displayed by this view.
- The view begins by using the Product model to retrieve all of the products from the database.
- The variables query, categories, sort, and direction are all initialised to None. Search queries, chosen categories, sorting options, and sorting direction will all be stored in these variables, in that order.
- The view then determines whether the request contains any GET parameters to search through.
- The view sorts of data according to the specified choice if a sorting option is provided in the GET parameters. It can sort by name, category, or any other available information.
- When a category is chosen in the GET parameters, only the products that fall within that category are displayed in the view.
- The view filters the products depending on the search query if one is provided in the GET parameters. The product names and descriptions are compared to the search query using case-insensitive partial matching.
- The current sorting choice and direction are stored in the variable current_sorting, which will be utilised in the template for display reasons.
- The filtered products, search terms, chosen categories, and current sorting data are used to generate the context dictionary.
- The view then renders the 'products/products.html' template using the supplied context.

Context processor:

- A context processor is a function that makes data available for use in all Django application templates. bag_contents: Calculations and information about a shopping bag or cart are handled by this context processor.
- It initialises variables for product count, total price, total price of bag components, and more.
- The bag data, which includes the products and their quantities, is retrieved from the session.

- By multiplying the number of each item by its corresponding product price, it goes through the things in the bag one by one and determines the overall cost.
- The price and number of products for each size are computed iteratively if an item in the bag has different sizes.
- Based on the entire cost, it assesses if the user qualifies for free delivery and, if so, computes the delivery fee.
- It computes the grand_total, which accounts for both the delivery fee and the overall cost.
- All of the calculated data is used to generate the context dictionary, which is then used as context in all application templates.

Cart Management:

Checkout:

Payment Processing using Stripes API:

The payment process is covered in the checkout.html and confirmed in the checkout_success.html files. The process is handled in the checkout views file.

cache_checkout_data view:

- Gets payment information (such as the client's secret) from the client side via a POST request.
- Use Stripes API to adjust the payment intent with user preferences and shopping bag-related metadata (such as save_info and username).
- returns an HTTP response with a 200 success status code or a 400 error status code.

checkout view:

- It creates a form to collect customer information and a Stripe card element for safe payment entry on the checkout page.
- If the user submits the form, a new Stripe payment intent is created depending on the total cost of the bag.
- checks the form's data for accuracy before creating an Order object and saves it directly to the database.
- checkout view:
- Based on the contents of the bag, and then creates OrderLineItem objects and links them to the order.
- The order is deleted and an error message is displayed if any item in the bag cannot be found.
- After a successful payment, the page redirects to the checkout_success view while taking the order number as input.

Signals:

Signals:

Functions known as signal handlers are called when specific events occur in the application. When an event occurs, they are utilised to carry out extra tasks or update relevant data. The application implements two signal handlers for the OrderLineItem model utilising the post_save and post_delete signals.

```python
from django.db.models.signals import post_save, post_delete
from django.dispatch import receiver
from .models import OrderLineItem

# exectues the update_on_save method
@receiver(post_save, sender=OrderLineItem)
# fucniton to handle signals from post save event
def update_on_save(sender, instance, created, **kwargs):
    """
    Update order total on lineitem update/create
    """
    instance.order.update_total()

# exectues the update_on_save method
@receiver(post_delete, sender=OrderLineItem)
# fucniton to handle signals from post save event
def update_on_delete(sender, instance, **kwargs):
    """
    Update order total on lineitem delete
    """
    instance.order.update_total()
```

Imports

from django.db.models.signals import post_save, post_delete:

- To handle post-save certain Django imports are used.

from django.dispatch import receiver:

- This import connects signal handlers to signal sender functions via the decorator function receiver.

from .models import OrderLineItem:

- It imports the OrderLineItem model from the same application. Individual line items in an order are typically represented by this model.

update_on_save:

- The OrderLineItem model's post_save signal is attached to the signal handler procedure called update_on_save.
- This function is linked to the OrderLineItem model's post_save signal by the @receiver(post_save, sender=OrderLineItem) decorator.
- Every time an existing OrderLineItem instance is modified or a new instance is generated like a save operation, the function is called.
- When this signal handler is invoked, it calls the corresponding Order instance's update_total() method to update the order total, which is the sum of the prices of all the line items in the order.

update_on_delete:

- The OrderLineItem model's post_delete signal is related to this signal handler function.

- This function is linked to the OrderLineItem model's post_delete signal via the @receiver(post_delete, sender=OrderLineItem) decorator.
- Every time an OrderLineItem instance is deleted, the function is invoked.
- When this signal handler is invoked, it runs the corresponding Order instance's update_total() method to update the order total following the deletion of the line item.

Fixtures:

Fixtures are used in JSON, XML, or YAML file formats. They serve as an image of all the data in your application, including model instances and the data they are linked to. Fixtures can be used to populate data for a variety of purposes, including setting up sample material, pre-loading reference data, and creating default user accounts. The folder is found in the application of the product in the fixtures folder.

In this application, they are used to load the product categories and product details using a JSON file.

Categories:

The categories.json are represented as a list of product categories in JSON format together with the corresponding data. A primary key (pk), a model type (model), and a set of fields with the category name (name) and a friendly name (friendly_name) are all present for each category.

To load the categories the command line command is used. “python3 manage.py loaddata categories”

Here is an example of the first category loaded below.

[{"pk": 1, "model": "products.category", "fields": {"name": "vitamins", "friendly_name": "Vitamins"}}]

Products:

The products.json is a JSON representation that contains data about the application's products. A primary key (pk), a model type (model), and some fields with information on the product—including its SKU, name, description, price, category, rating, and image—are included in each entry for a given product. this helps preload the database with the product information.

To load the products the command line command is used. “python3 manage.py loaddata products”

Here is an example of the first product loaded below.

"pk": 1,

"model": "products.product",

"fields": {

"sku": "p000000001",

"name": "Beeline Multivitamins Tablets",

"description": "Beeline Multivitamin Tablets contain our most advanced vitamin and mineral formula to provide comprehensive nutritional support. Suitable for vegetarians, the supplement supports health, vitality, wellbeing and energy levels.",

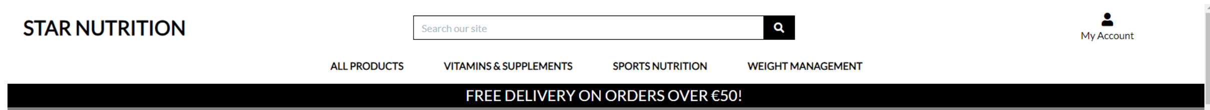
"price": 6.80,

```
            "category": 1,
            "rating": 4.4,
            "image": "multi.jpg"
        }
    },
```


2.4. Graphical User Interface (GUI)

Main Navbar:

The main navbar is available on all pages of the application.

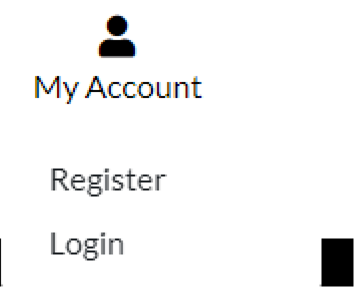


On the top left-hand side of the page, the Star Nutrition logo is present as a clickable logo.

STAR NUTRITION

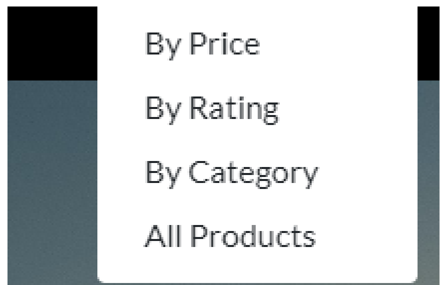
In the centre of the page, there is a search bar that will return any product once the search relates to the name, key word, category etc.

In the top right-hand side of the page, there is my account icon which is a clickable button that is a dropdown menu, allowing a user to register or log in if logged in the user can logout or go to their profile page to update their orders.



In the centre below the search bar, the applications dropdown menus the first dropdown menu is for “ALL PRODUCTS”. When clicked the dropdown menu shows 4 options.

All Products:



By Price:

Sorts products and displays by price in an ascending order, when clicked.

By Rating:

Sorts products and displays by rating in a descending order, when clicked.

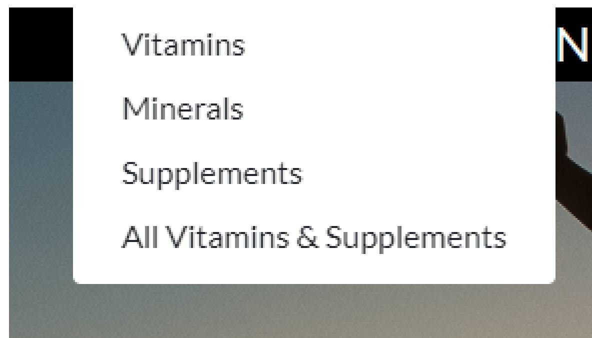
By Category:

Sorts products and displays by category in an ascending order, when clicked.

By Products:

Displays all products when clicked.

Vitamins & Supplements:



Vitamins:

Sorts products and displays by category vitamins, when clicked.

Minerals:

Sorts products and displays by category Minerals, when clicked.

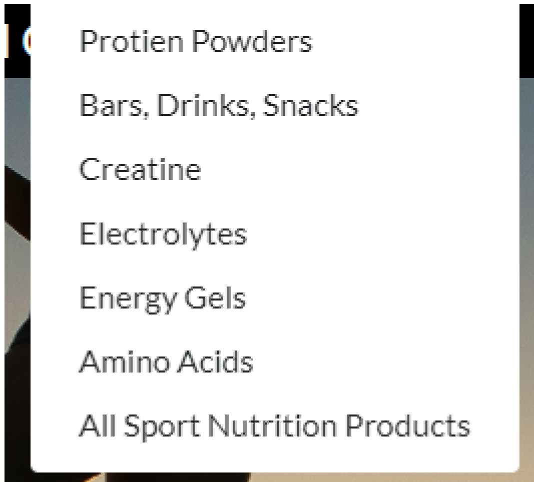
Supplements:

Sorts products and displays by category Supplements, when clicked.

All Vitamins & Supplements:

Sorts products and displays by Vitamins and supplements, when clicked.

Sports Nutrition:



Protein Powders:

Sorts products and displays by category protein, when clicked.

Bars, Drinks, Snacks:

Sorts products and displays by category Bars, Drinks, and Snacks, when clicked.

Creatine:

Sorts products and displays by category Creatine, when clicked.

Electrolytes:

Sorts products and displays by category electrolytes, when clicked.

Energy Gels:

Sorts products and displays by category Energy Gels, when clicked.

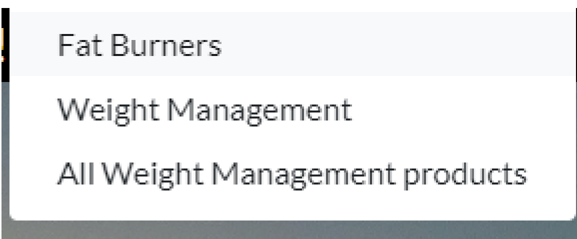
Amino Acids:

Sorts products and displays by category Amino Acids, when clicked.

All Sports Nutrition Products:

Sorts products and displays by Sports Nutrition, when clicked.

Weight Management:



Fat Burners:

Sorts products and displays by category Fat Burners, when clicked.

Weight Management:

Sorts products and displays by category Weight Management, when clicked.

All Weight Management products:

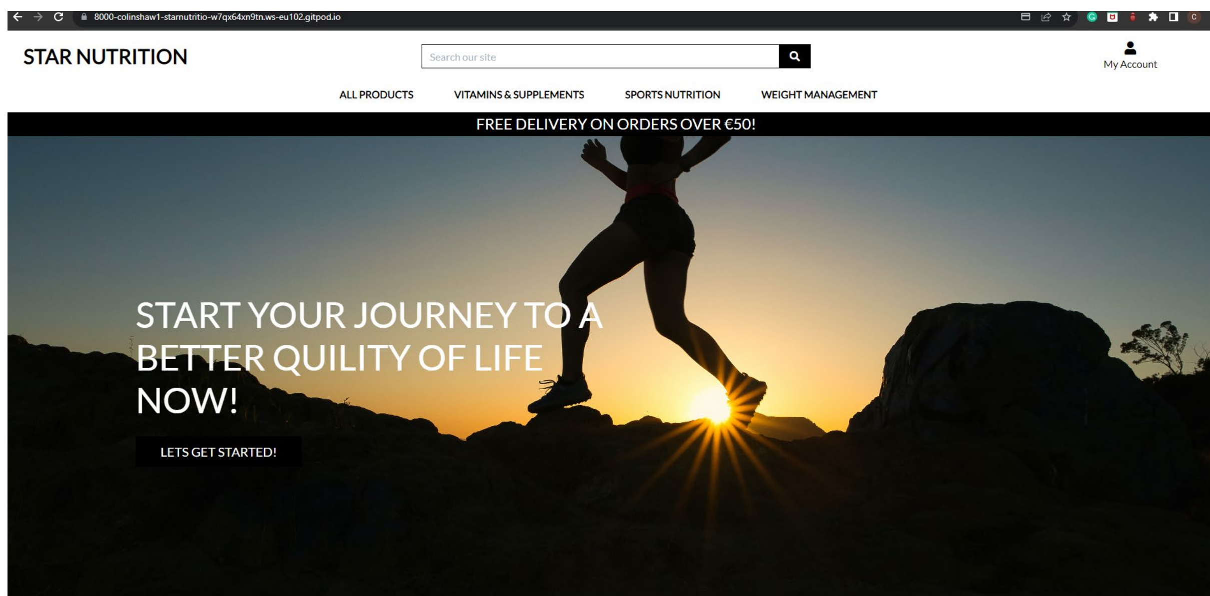
Sorts products and displays by weight management, when clicked.

Free Delivery on all orders over €50:

Displays the free delivery banners.

FREE DELIVERY ON ORDERS OVER €50!

Homepage:

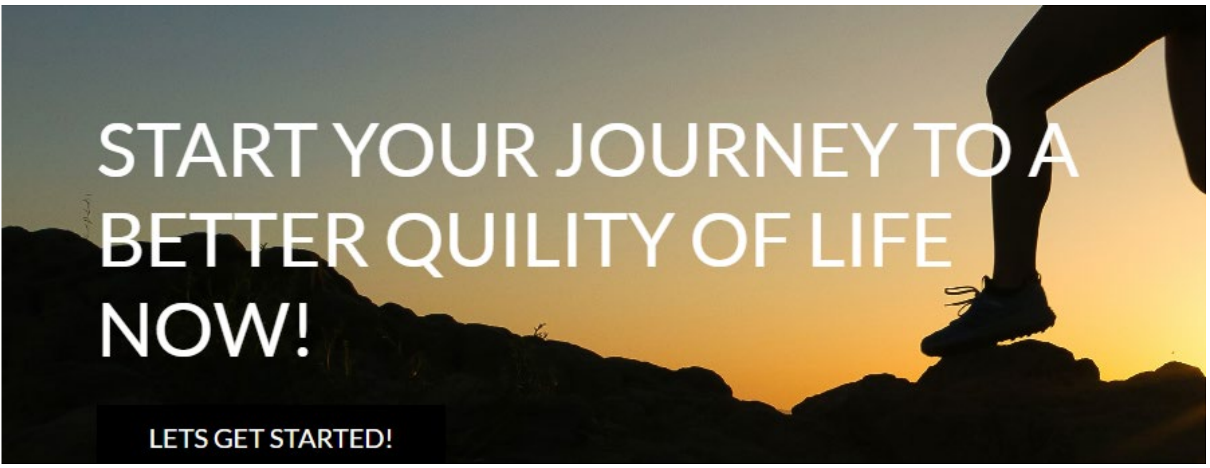


Star Nutrition’s home page serves as its landing page. To aid consumers in finding what they're looking for fast, it consists of all the platform's products, shopping bags, delivery information, a navbar, logo, user profiles and a search bar.

Applications main landing image.

This is an inspirational quote to try to engage users to browse products.

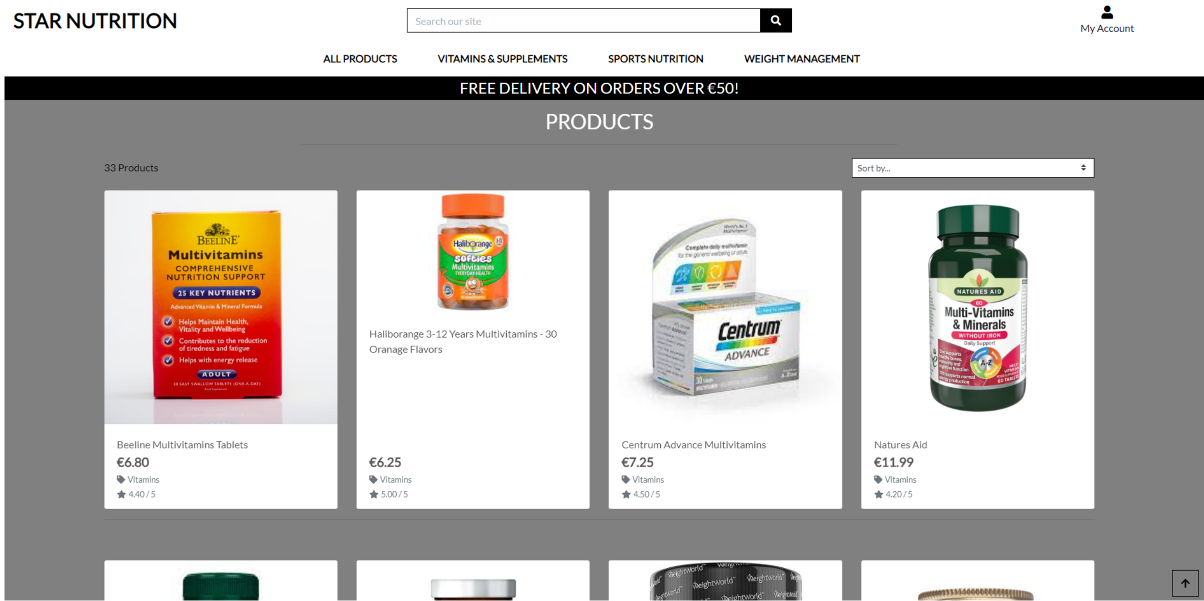
A let's get started button which brings the user to the products page when clicked.



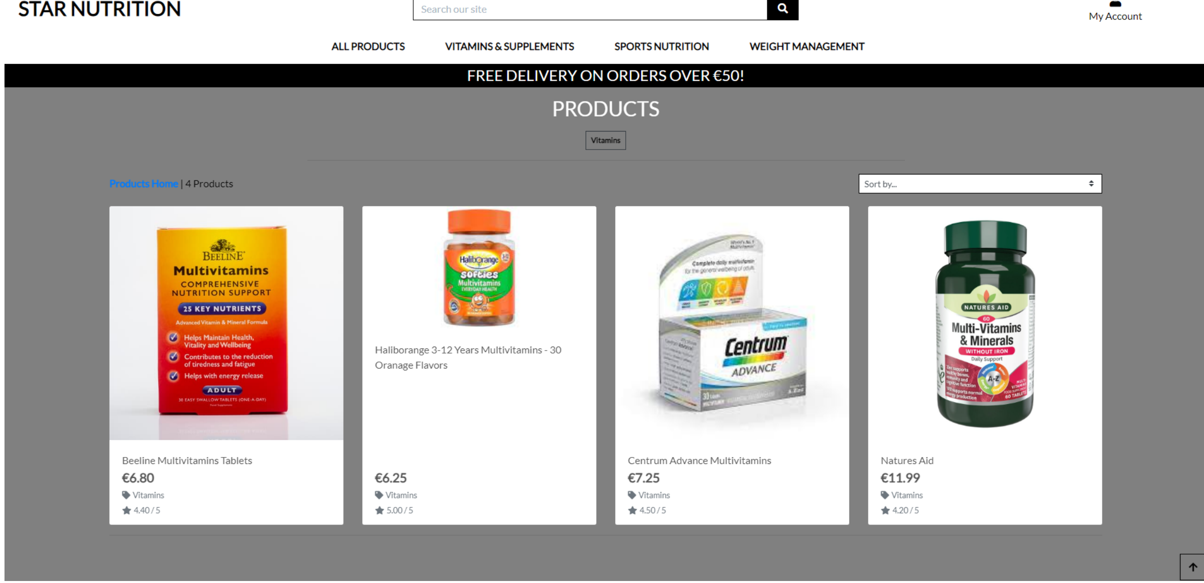
Products Page:

A list of products in a particular category is displayed on this page. Every product has a description, price, image, and "Add to Cart" button.

All products have been displayed.



Just a single category has been displayed. Vitamins in this instance.

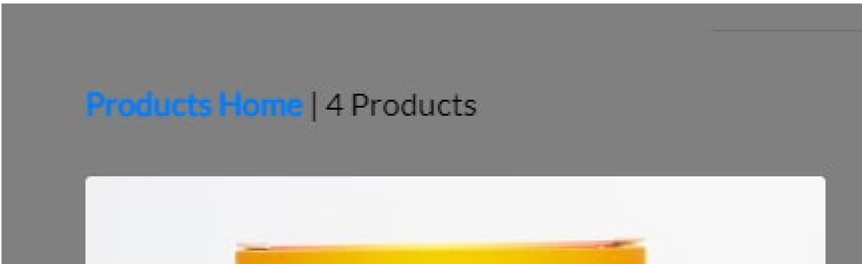


The products page has the navbar like all pages.

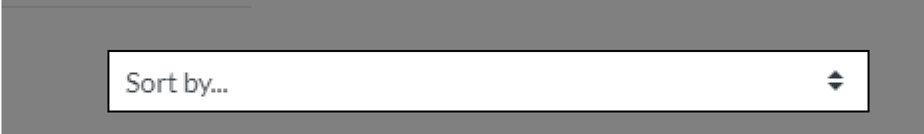
At the top left of the page, the number of products in the category you have selected has been displayed.

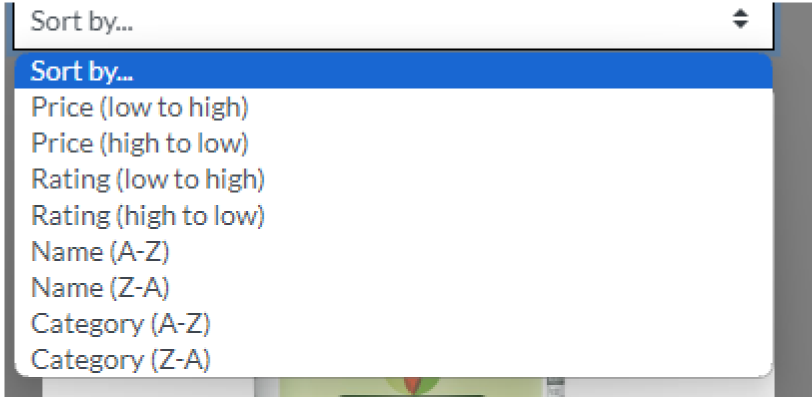
All products have been displayed.

Just vitamins show four products are being displayed and a button to return to the whole products page.



Sort by dropdown:





Price:

Users can sort products by the lowest price to highest or highest price to lowest.

Rating:

Users can sort products by the lowest rating to highest or highest rating to lowest.

Name:

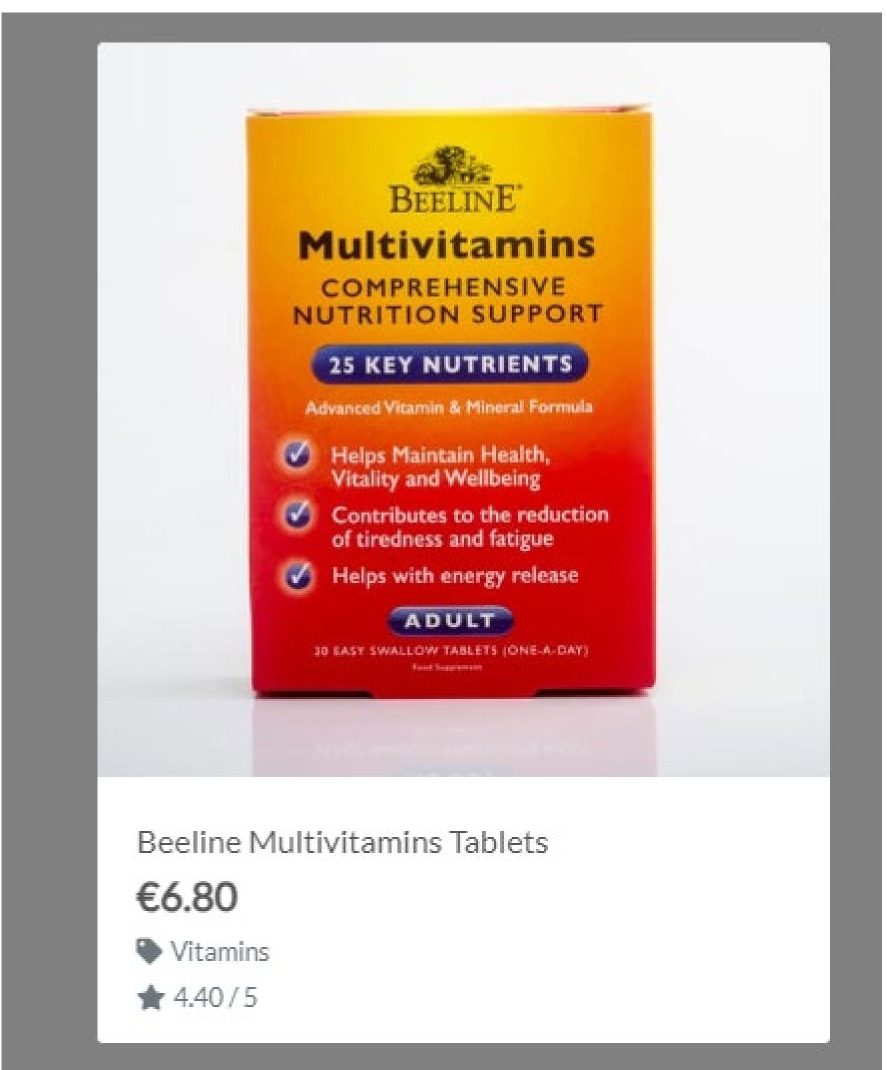
Users can sort products by name in alphabetical order from A-Z or in reverse Z-A.

Category:

Users can sort products by name in category order from A-Z or reverse Z-A.

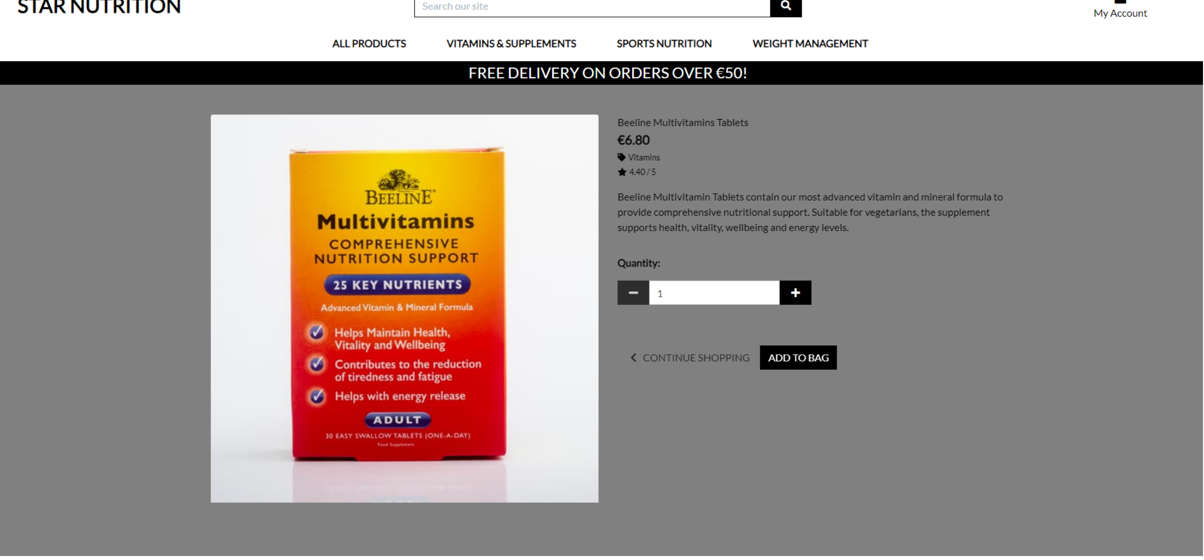
Products:

The products are displayed via an image, with the name, price, category and rating displayed.



Products Details Page:

The product detail bag has the nav bar the same as both other pages. This page displays the product's image, name, price, description, category, and rating. JavaScript button to add a certain quantity, a continue shopping button and an add to bag button.



Product Image:

Product Information:

The product information is displayed for users to read more about certain products.

Beeline Multivitamins Tablets

€6.80

Vitamins

4.40 / 5

Beeline Multivitamin Tablets contain our most advanced vitamin and mineral formula to provide comprehensive nutritional support. Suitable for vegetarians, the supplement supports health, vitality, wellbeing and energy levels.

Quantity selector:

Users select the quantity of products they want to add to their shopping cart.

Continue Shopping:

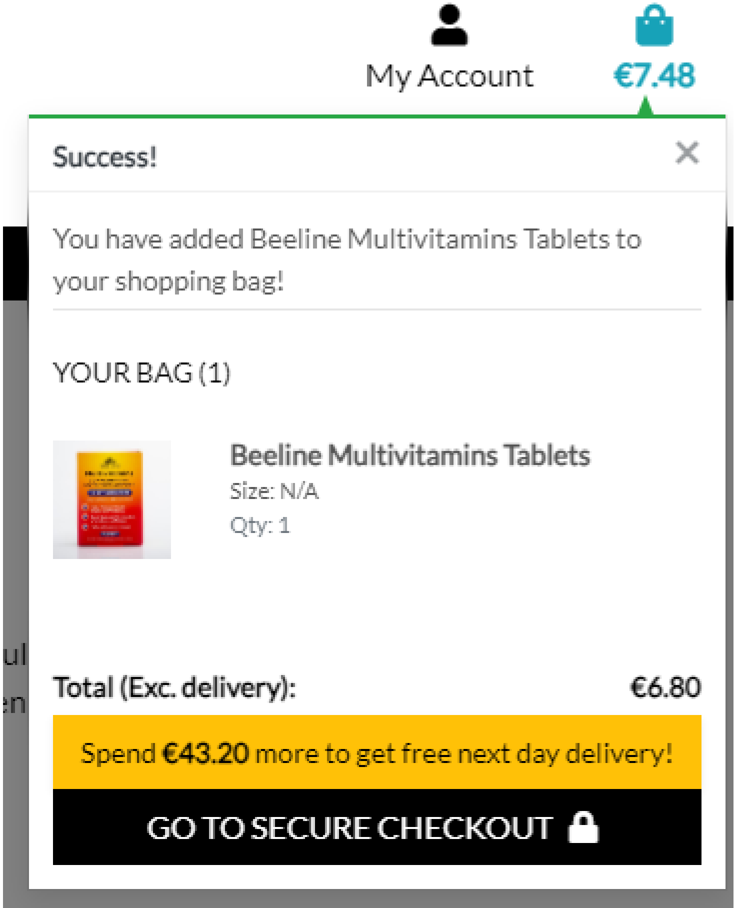
Users can select continue shopping after adding items to their bag or if they simply want to return to the products selection they were on before.

Add to Bag:

Users can click the add to bag button and this will add the items to their shopping cart.

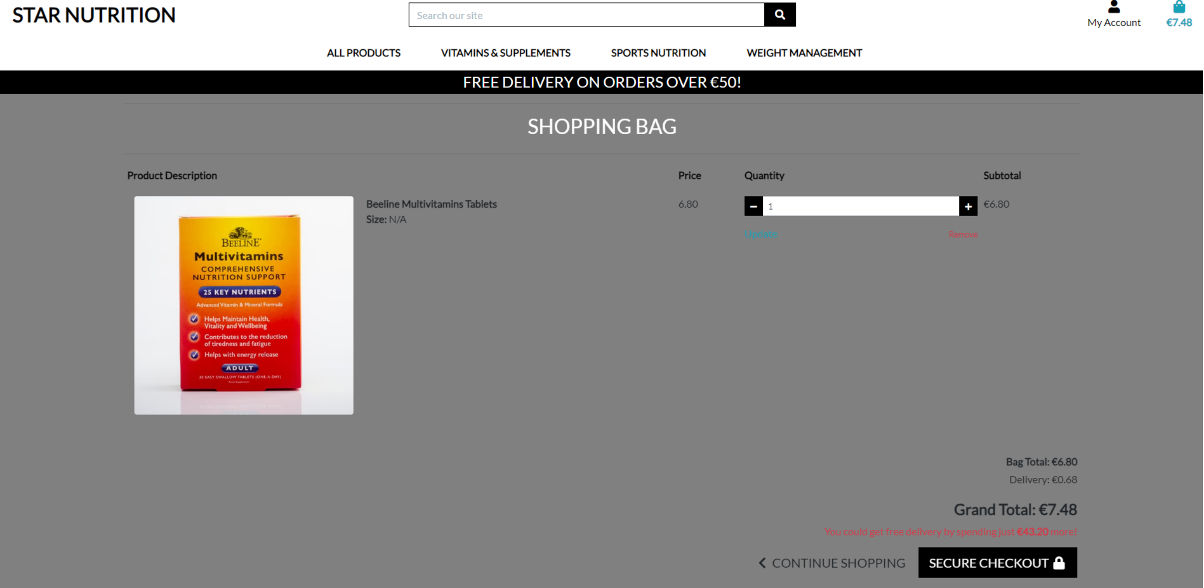
Successfully adding items to the bag:

When a user has selected the item they want to add to their shopping cart a success message will appear showing the items added to the cart and the shopping cart icon will appear.



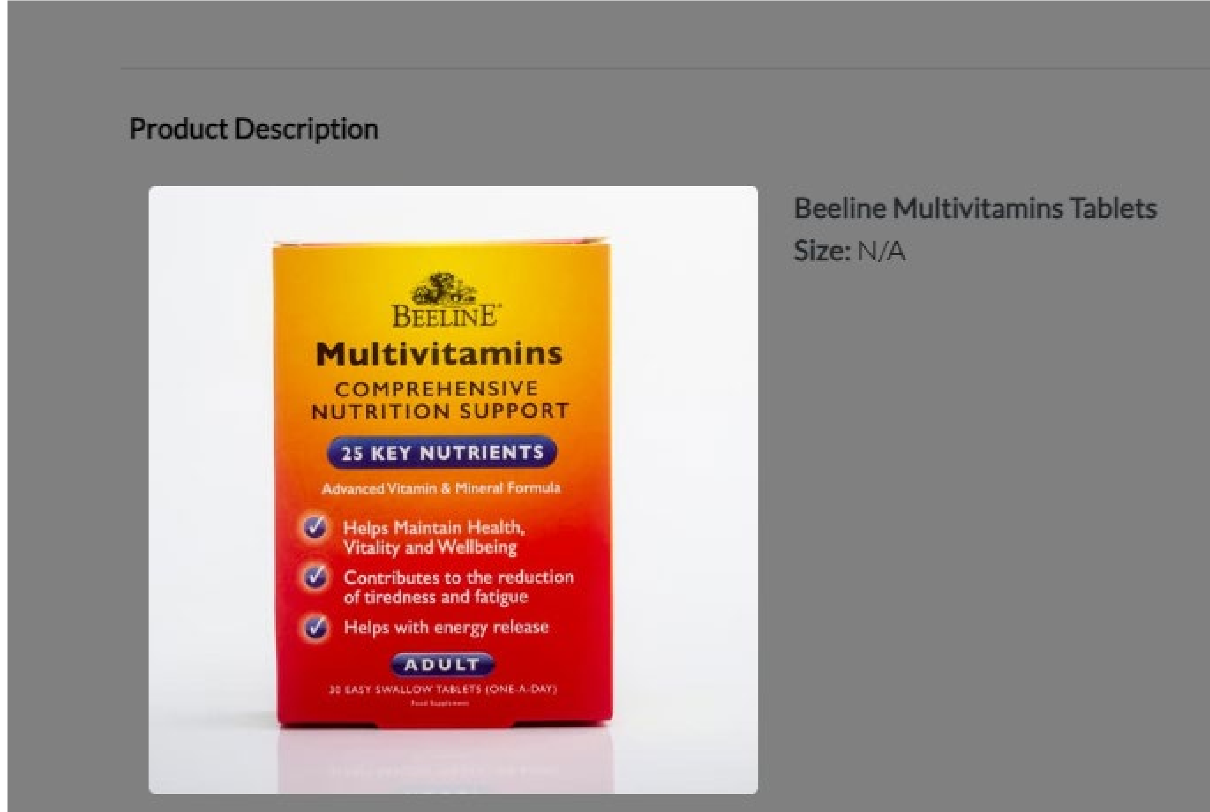
Shopping Cart:

The nav bar is present like other pages except this time with the cart present as items are in the shopping cart. The shopping cart page shows every item the user has added. It displays the name, quantity, cost, and total cost of every item in the cart, and delivery fee, continue shopping and secure shopping bag. Customers can change amounts or take things out of their shopping carts.



Product description:

Shows the product's image, and name and if the item has a size shows the size.



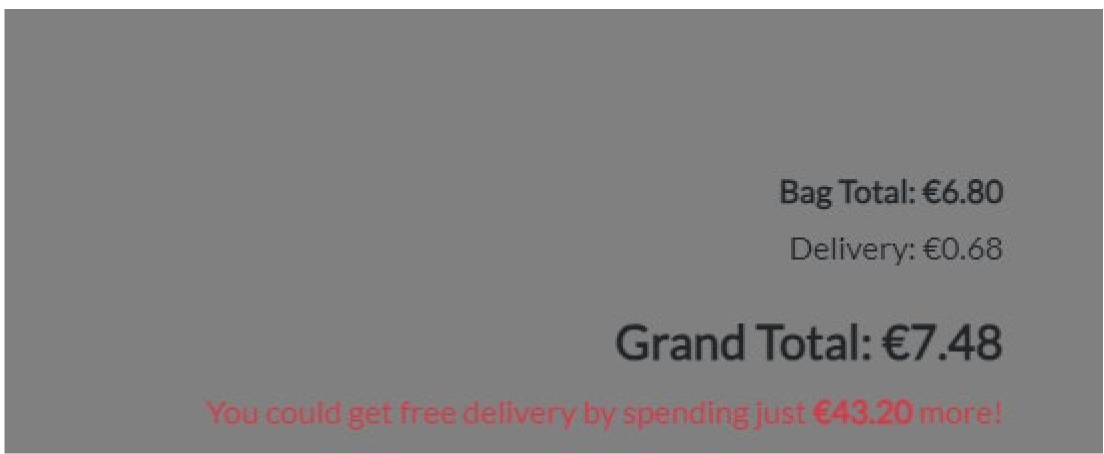
Item price:

Item quantity selector and subtotal:

A quantity selector is available to add or remove items from the card as well as display the subtotal.

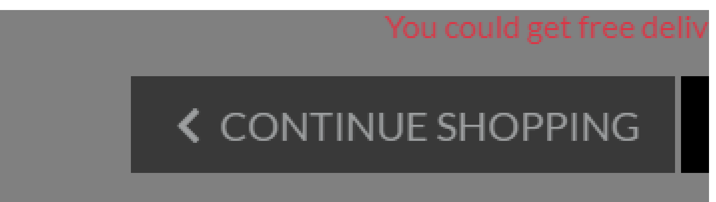
Shopping bag total:

The bag total is displayed, the delivery cost and the total with delivery charges. A message is displayed to show the user if the spend a certain amount they will get free delivery.



Continue Shopping:

A button to bring the user bag to the products bag to continue shopping.

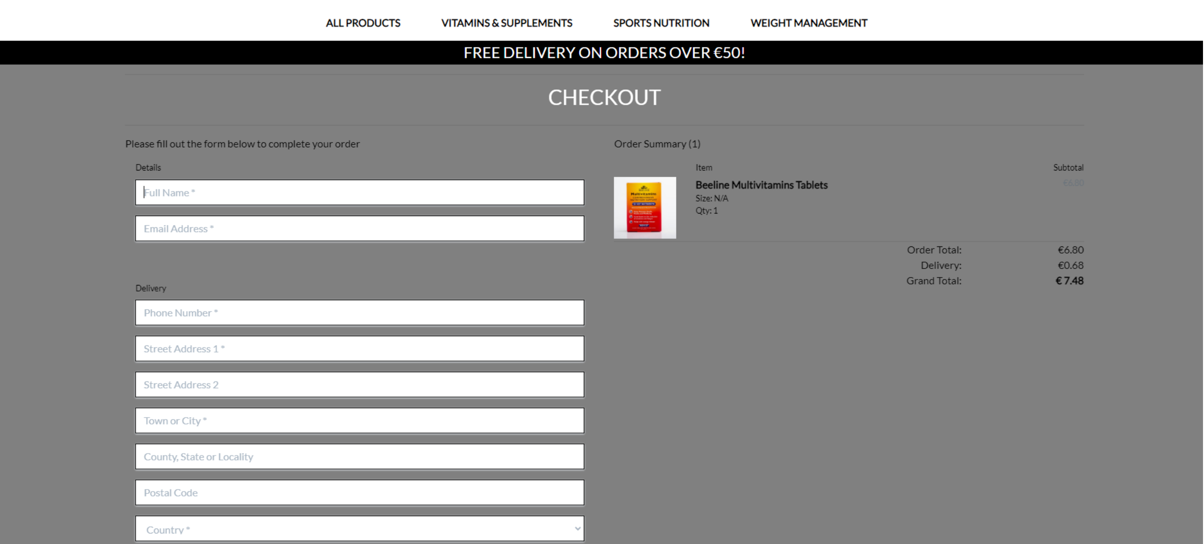


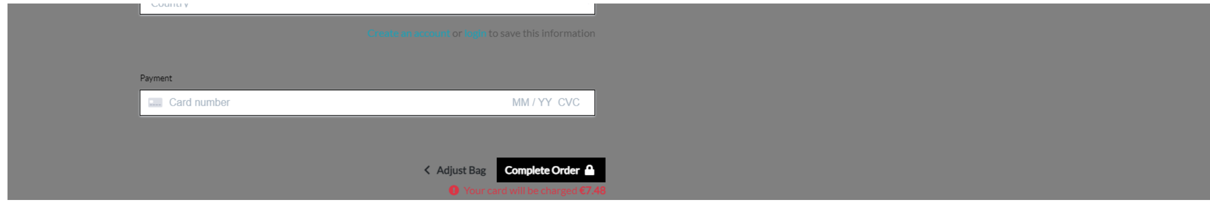
Secure Checkout:

A button to securely checkout this button will bring the user to the checkout bag to enter their details.

Checkout:

Users enter their shipping and payment details on the checkout page. Before going to the payment gateway, they can review their order.





Complete the order form:

Users enter the details such as name, email, and delivery detail:

Please fill out the form below to complete your order

Details

Full Name *

Email Address *

Delivery

Phone Number *

Street Address 1 *

Street Address 2

Town or City *

County, State or Locality

Postal Code

Country *

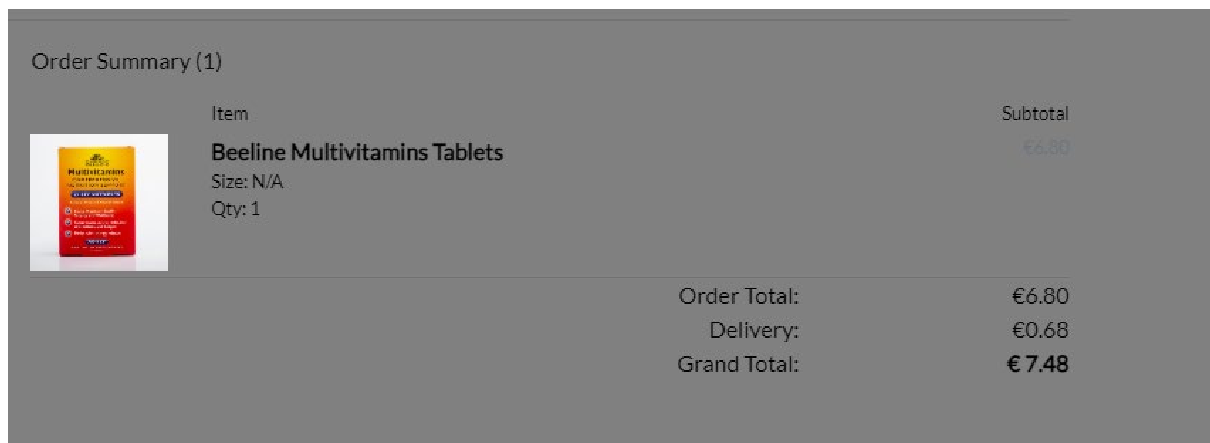
Create an account or login to save this information

Stripe Payment:

Users enter the card details into the stripe payment API.

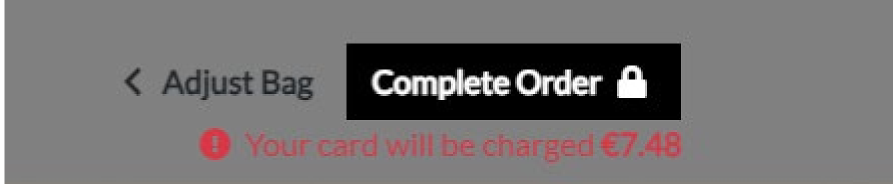
Order Summary:

The order summary so users can check all is in order before checking out.



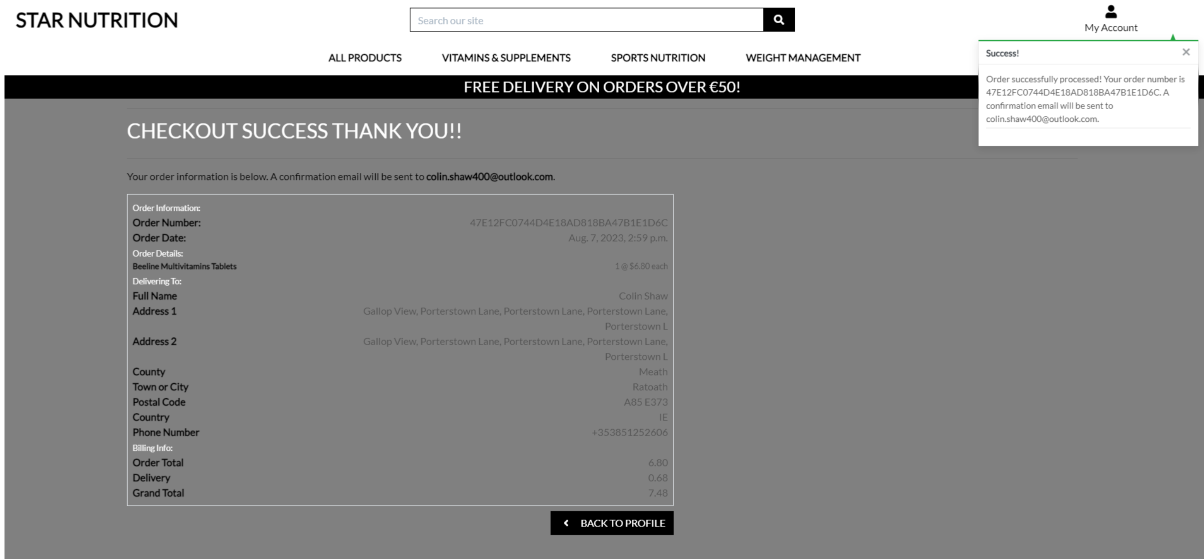
Complete order:

Users can adjust their bags by clicking the adjust bag button. Complete the order by clicking the order button and will be brought to the checkout success page. The red warning information tells the user their card will be charged.



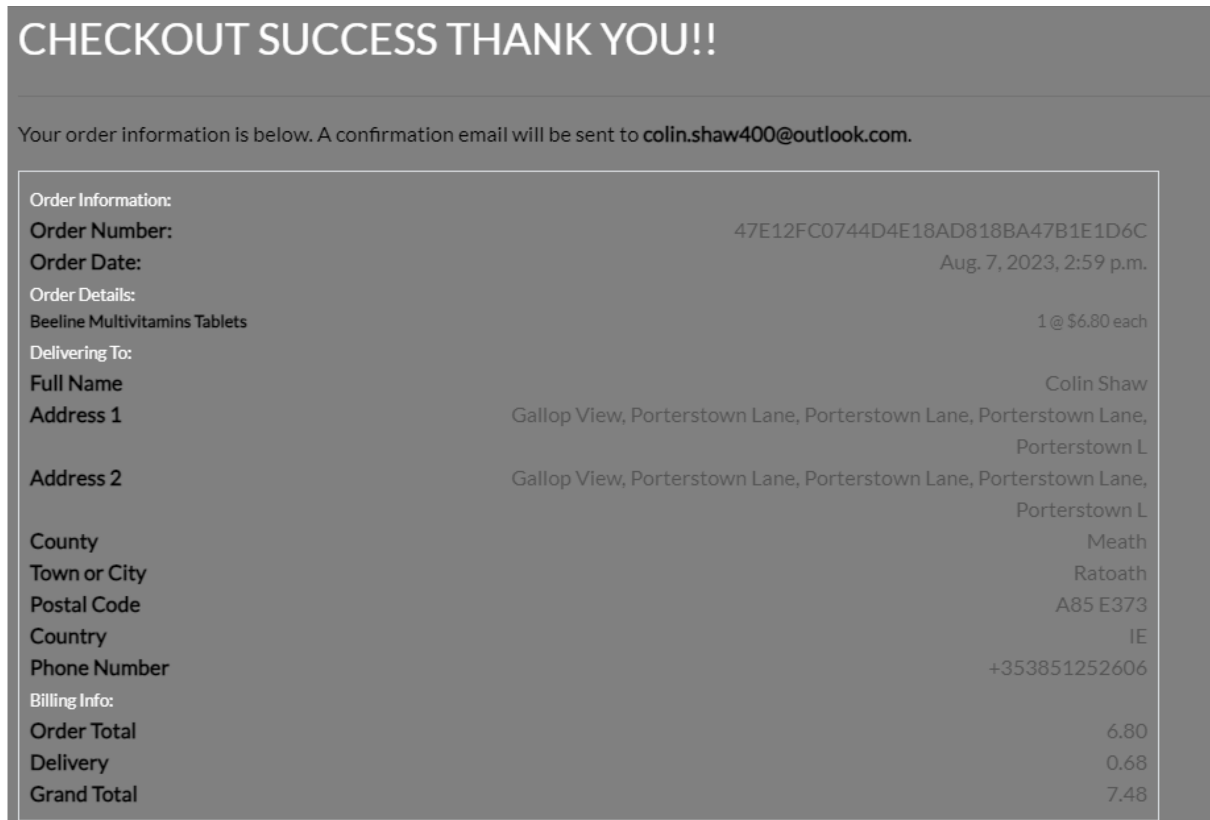
Checkout Succes:

The nav bar is present like other pages. The checkout success page shows users what they have purchased along with their details. It displays the order information, details, delivery address and billing info and a back-to-profile button and a success message with the order number.



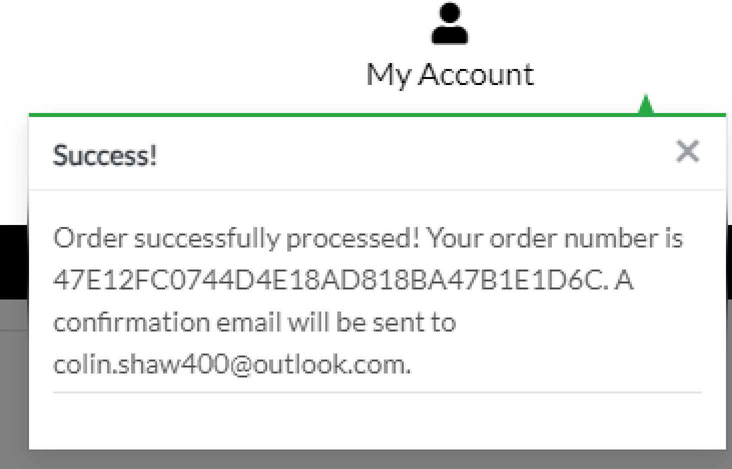
Order details:

The user is showing the order information, order details, delivery address and billing information.



Order Success:

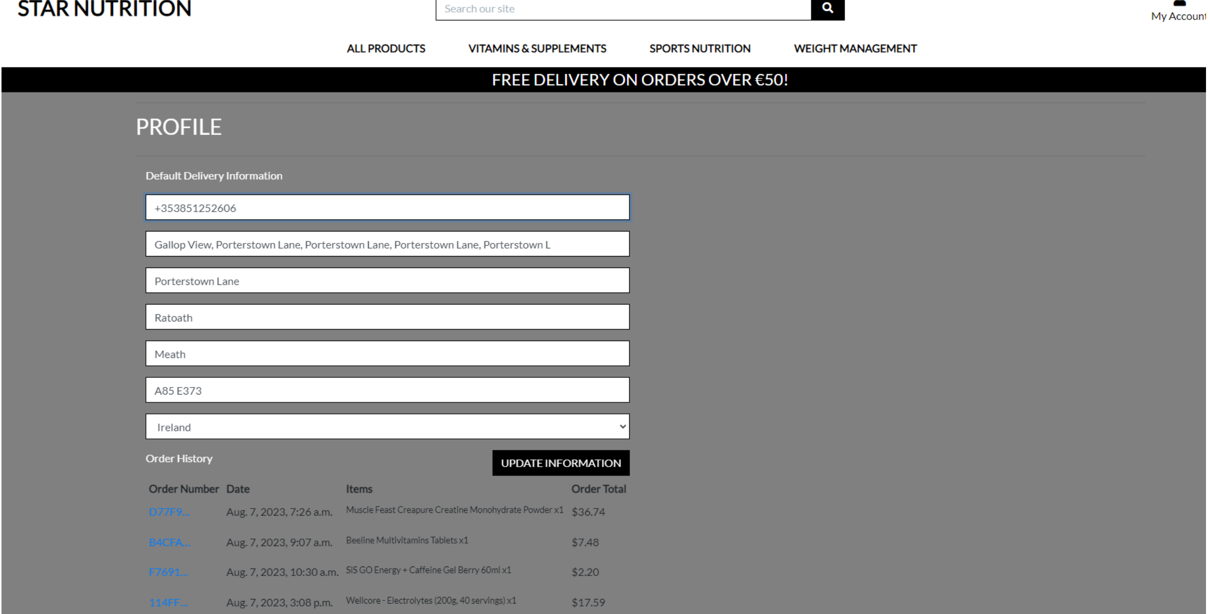
An order success message is showing and the order number.



Back to Profile:

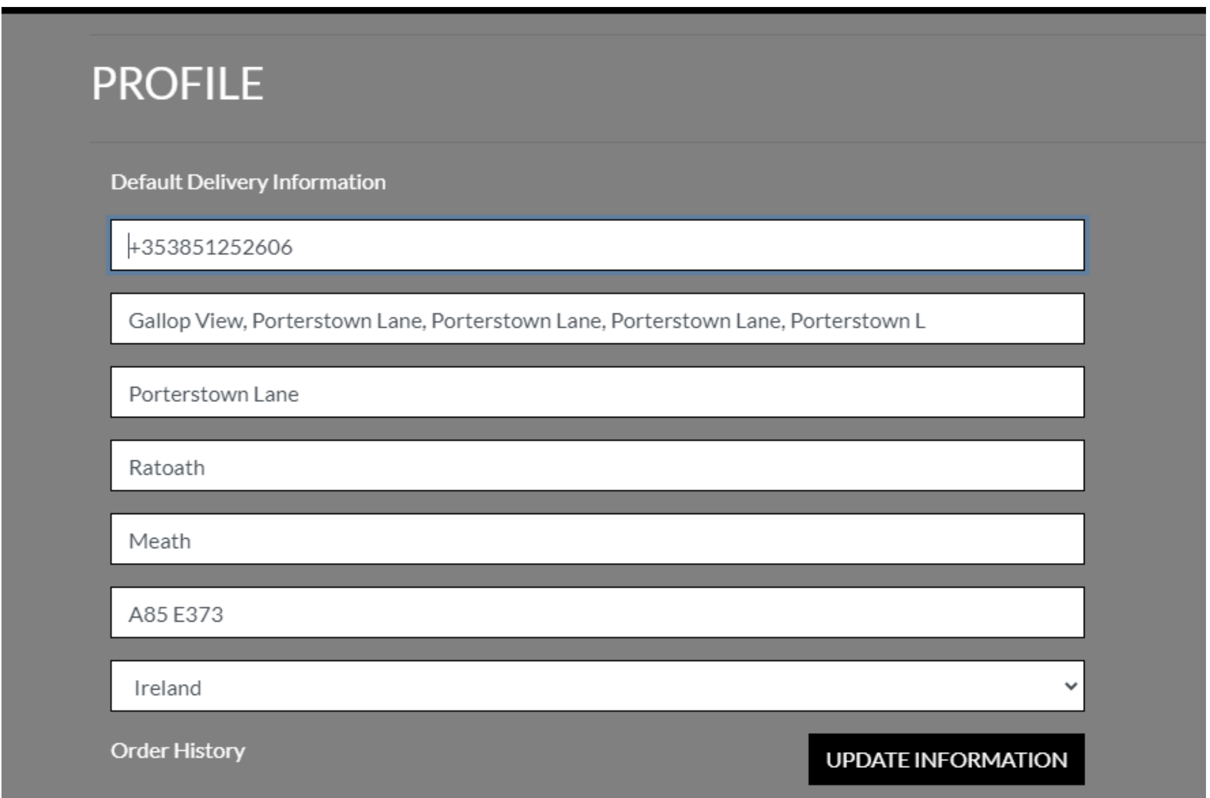
Profile Page:

The profile page has the navbar like all other pages. This is where users can update their details, like number address, number etc. The user can view previous orders also.



Delivery information:

Users can update their delivery information in this form.



Order history:

Users can check previous order details.

Order History

UPDATE INFORMATION

Order Number	Date	Items	Order Total
D77F9...	Aug. 7, 2023, 7:26 a.m.	Muscle Feast Creapure Creatine Monohydrate Powder x1	$36.74
B4CFA...	Aug. 7, 2023, 9:07 a.m.	Beeline Multivitamins Tablets x1	$7.48
F7691...	Aug. 7, 2023, 10:30 a.m.	SiS GO Energy + Caffeine Gel Berry 60ml x1	$2.20
114FF...	Aug. 7, 2023, 3:08 p.m.	Wellcore - Electrolytes (200g, 40 servings) x1	$17.59

2.5. Testing

First Time User

a. As a user visiting the site for the first time you want to be able to understand the main objective of the application.

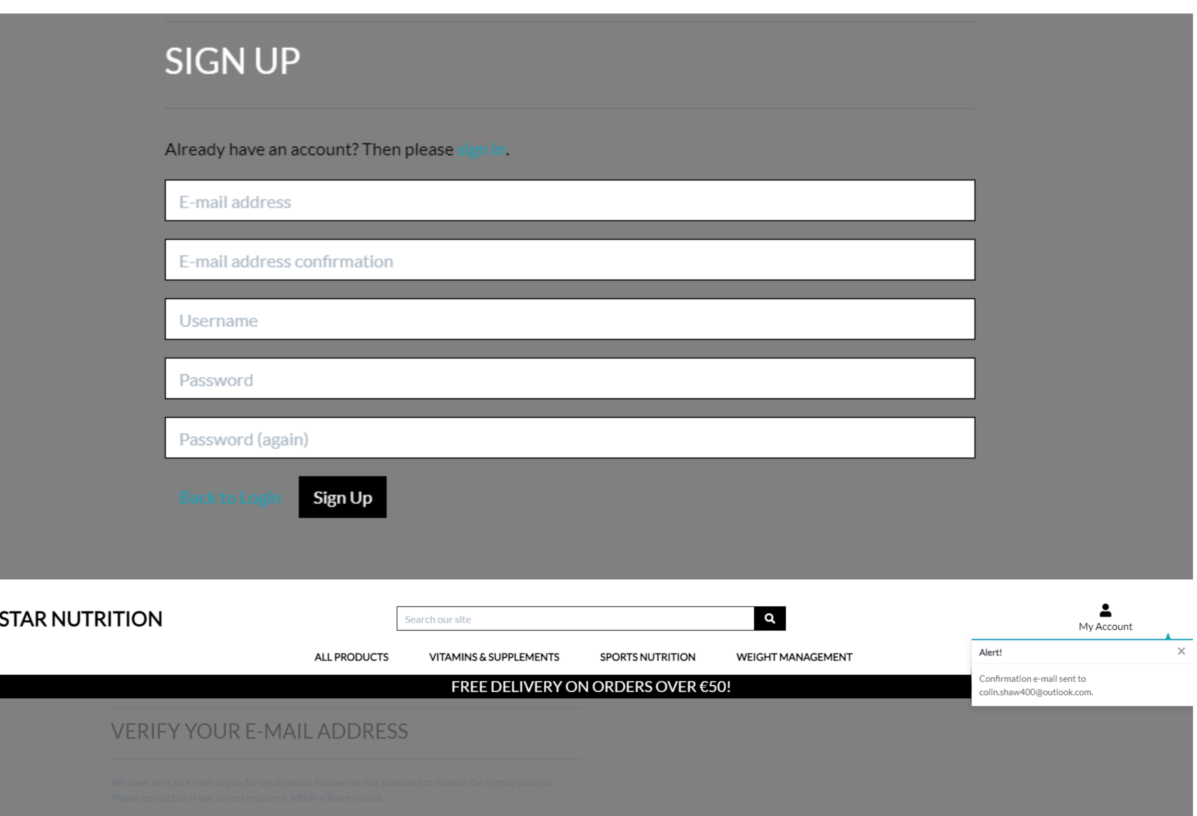
• This is achieved and a user can easily find the way through the application.

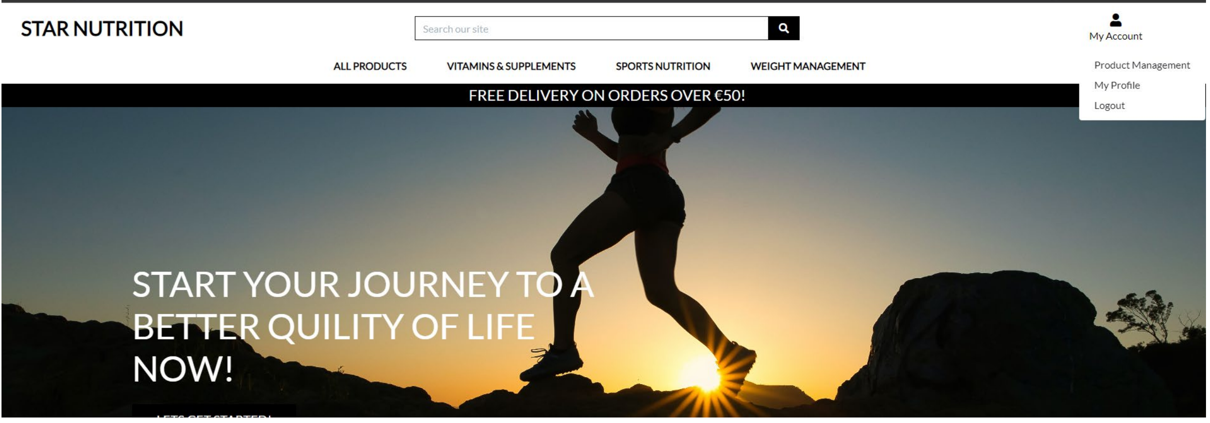
b. As a user visiting the site for the first time you want to easily navigate the site.

• A user will navigate through and find products and register easily.

c. As a user visiting the site for the first time you want to be able to set up a profile.

• A user can register for a profile easily through the profile icon which allows a user to register or log in.



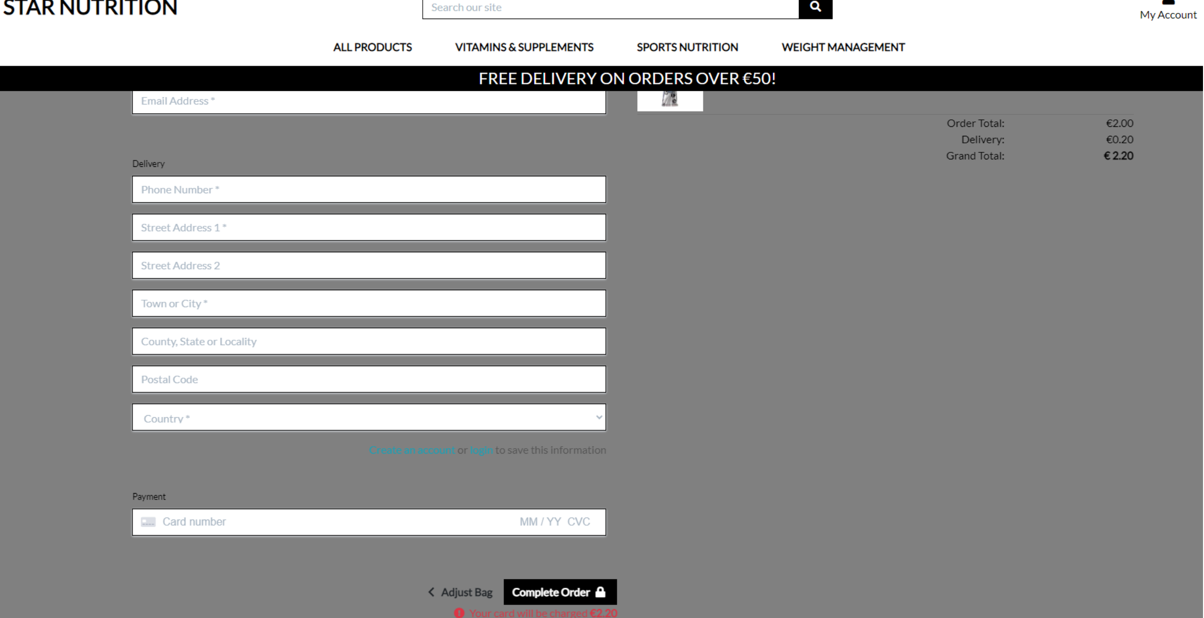


d. As a first-time user you want to be able to add items to your cart.

- As a first-time user, you can add items to your cart.

e. As a first-time user you want to be able to create an account at checkout.

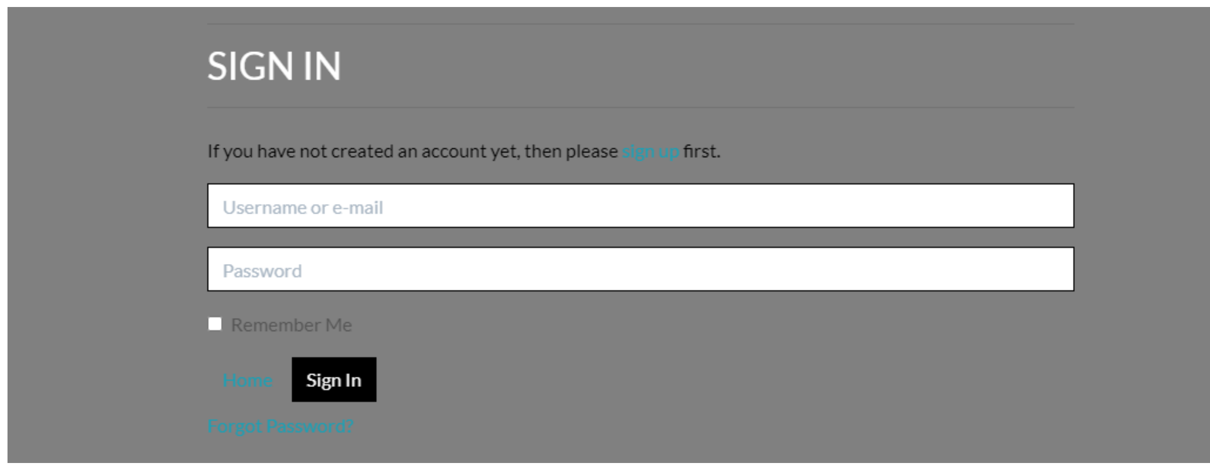
- As a first-time user, you can create an account while entering your details at the checkout page.

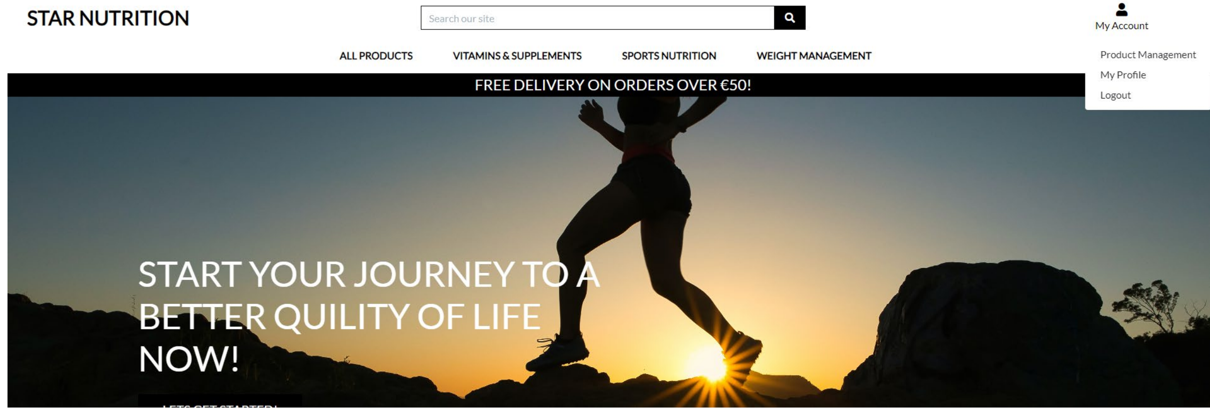


Frequent User

a. As a returning user you want to be able to log in to your profile.

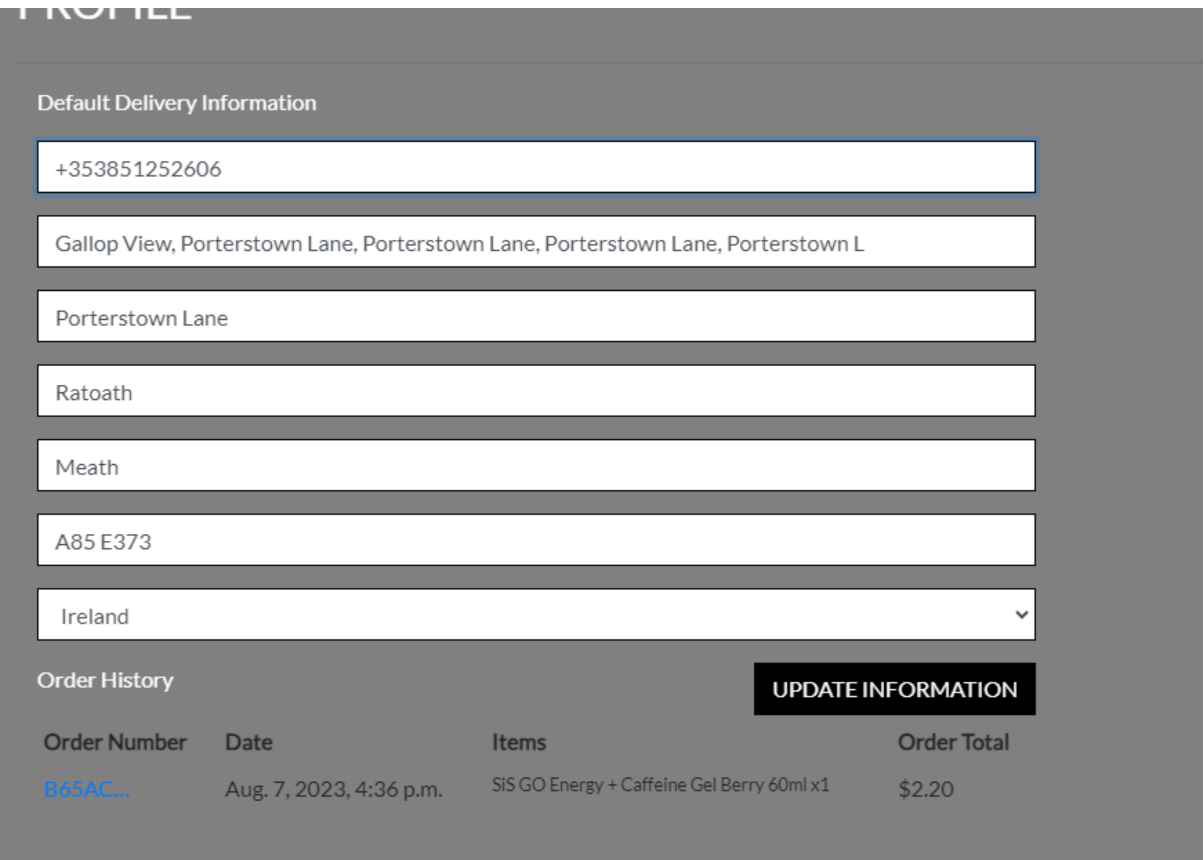
- A returning user can log in easily if they have a profile set up.





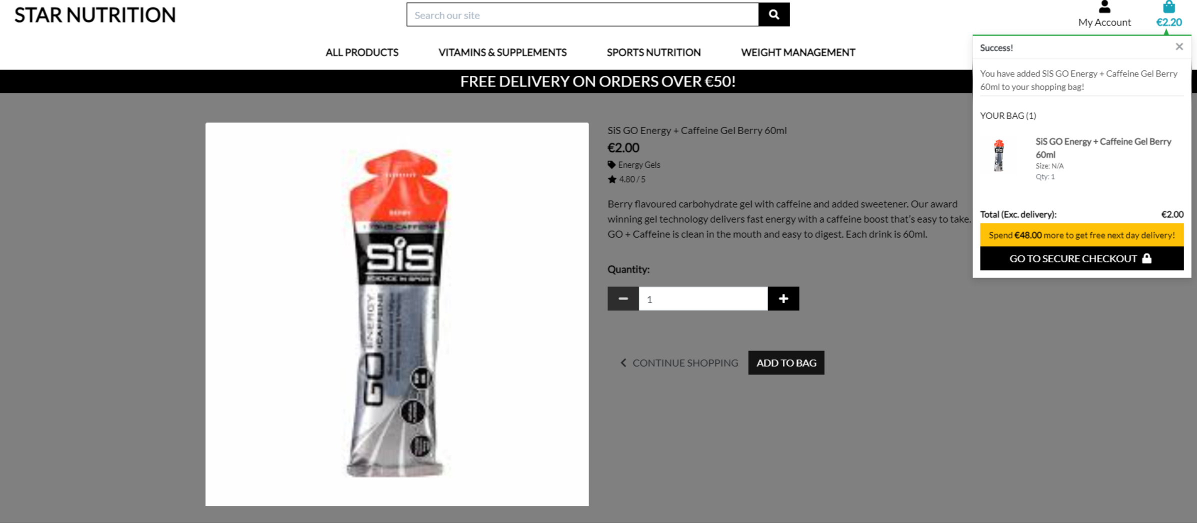
b. As a returning user you want to be able to edit your profile.

- A user can edit details or reset passwords easily.



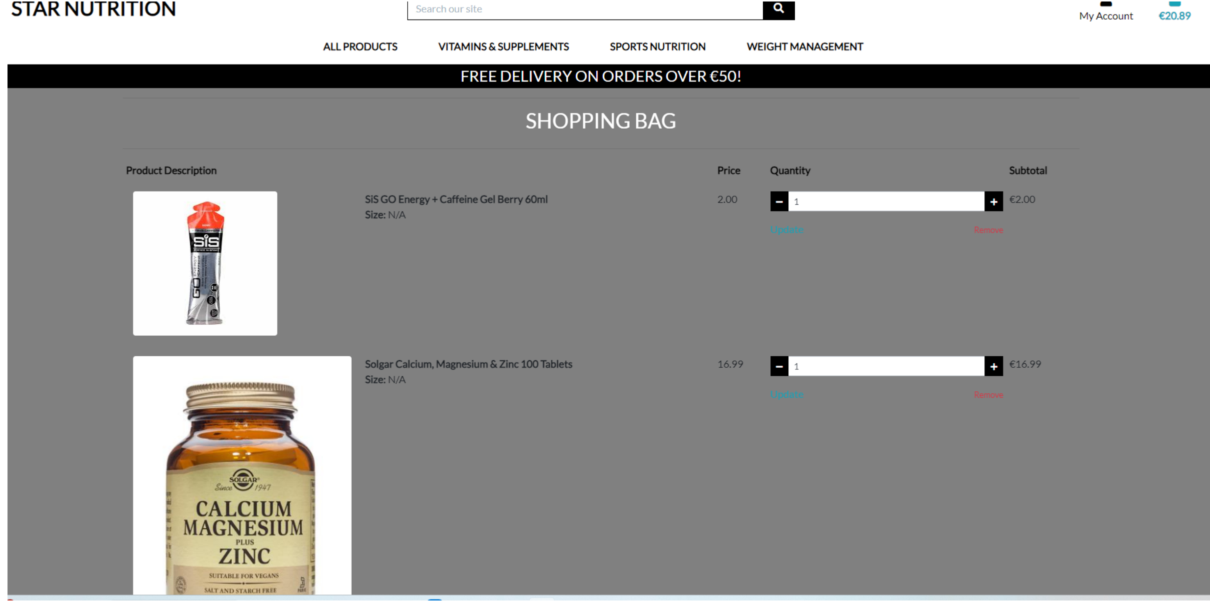
c. As a returning user you want to be able to add products to your cart.

- A user can products and a selected quantity to their cart.



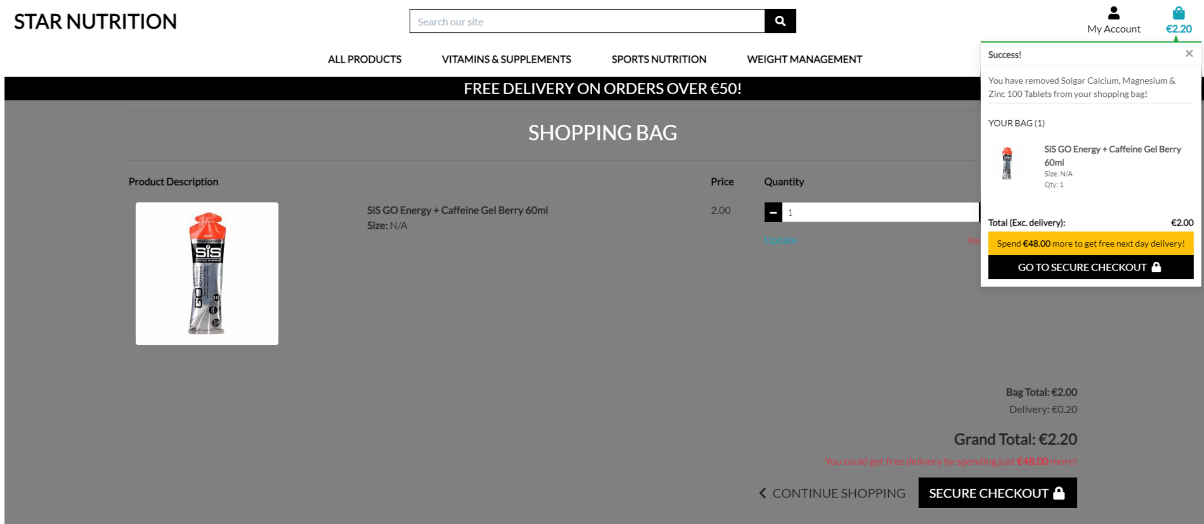
d. As a returning user you want to be able to check your cart.

- A user can check their cart through the cart icon and view and update their cart.



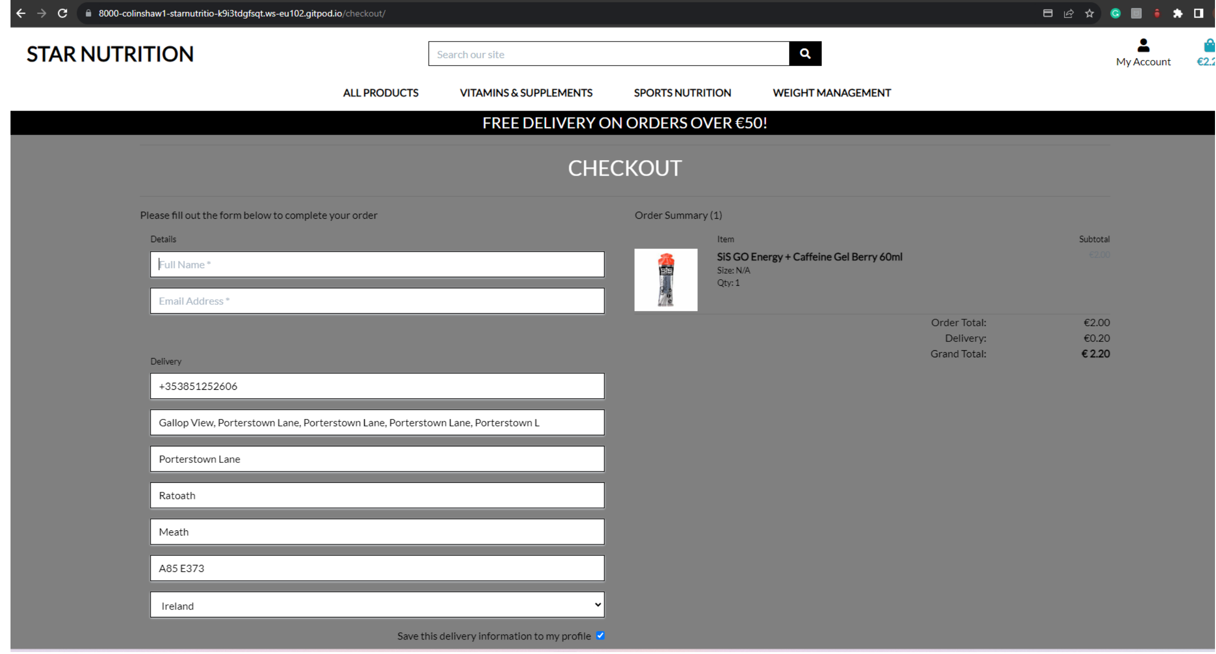
e. As a returning user you want to be able to edit your cart.

- a user can update the cart, and remove or update quantities of items.



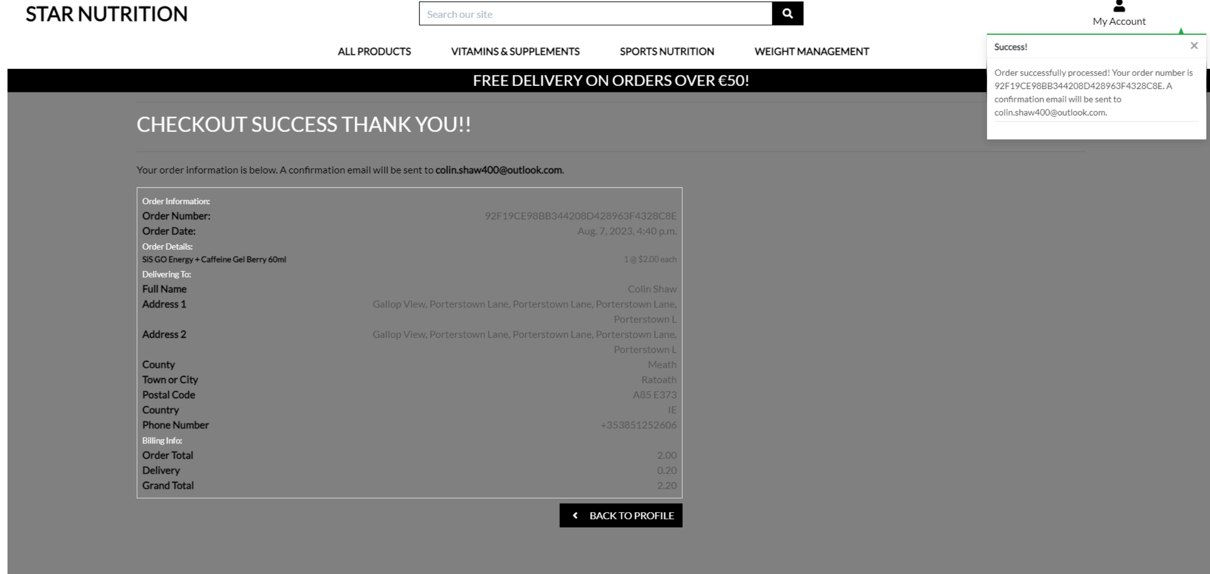
f. As a returning user you want to be able to check out your cart.

- A user can easily checkout and fill in their details for secure checkout.



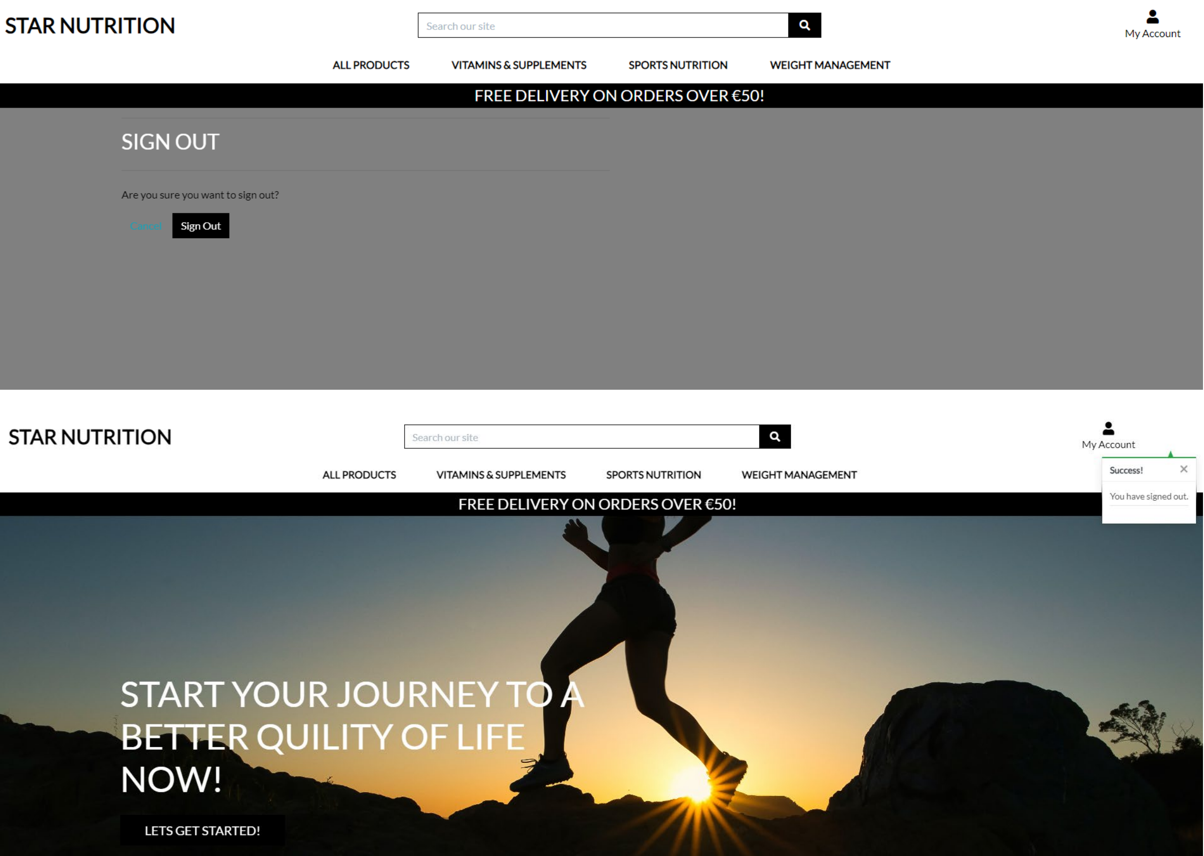
g. As a returning customer you want to be able to check out securely.

- As a returning customer, you can check out securely.



g. As a returning user you want to be able to log out.

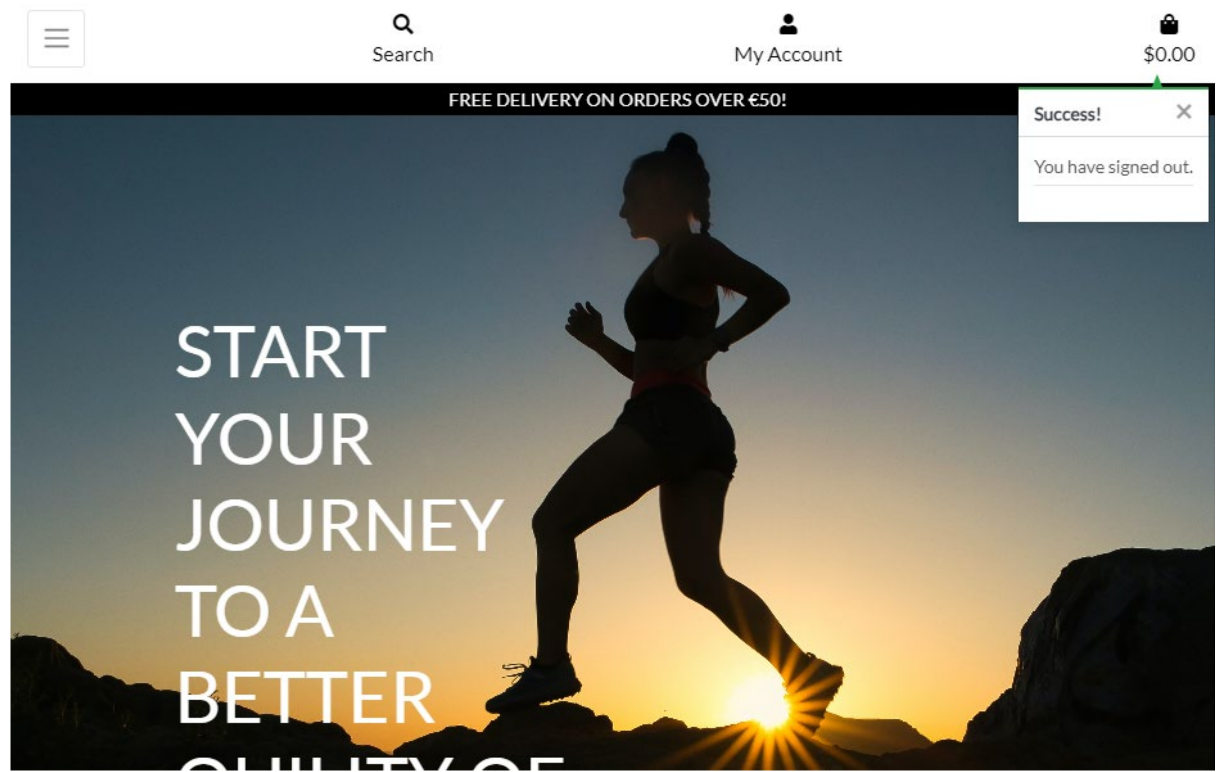
- A user can log out of their account once they want to end their session.



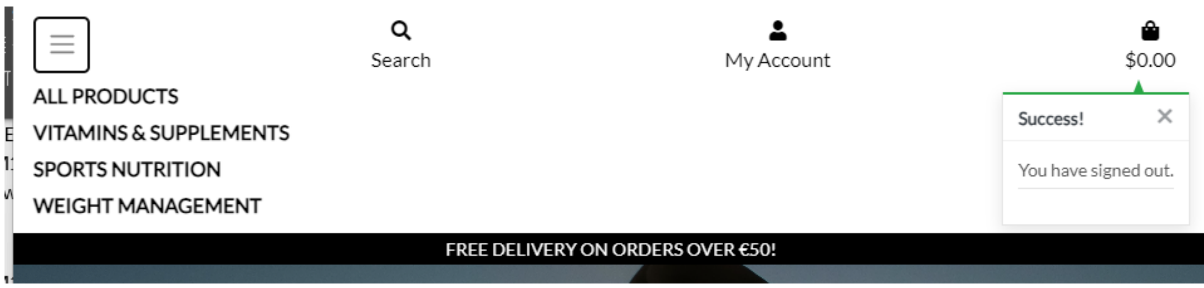
Testing:

Smaller screen testing.

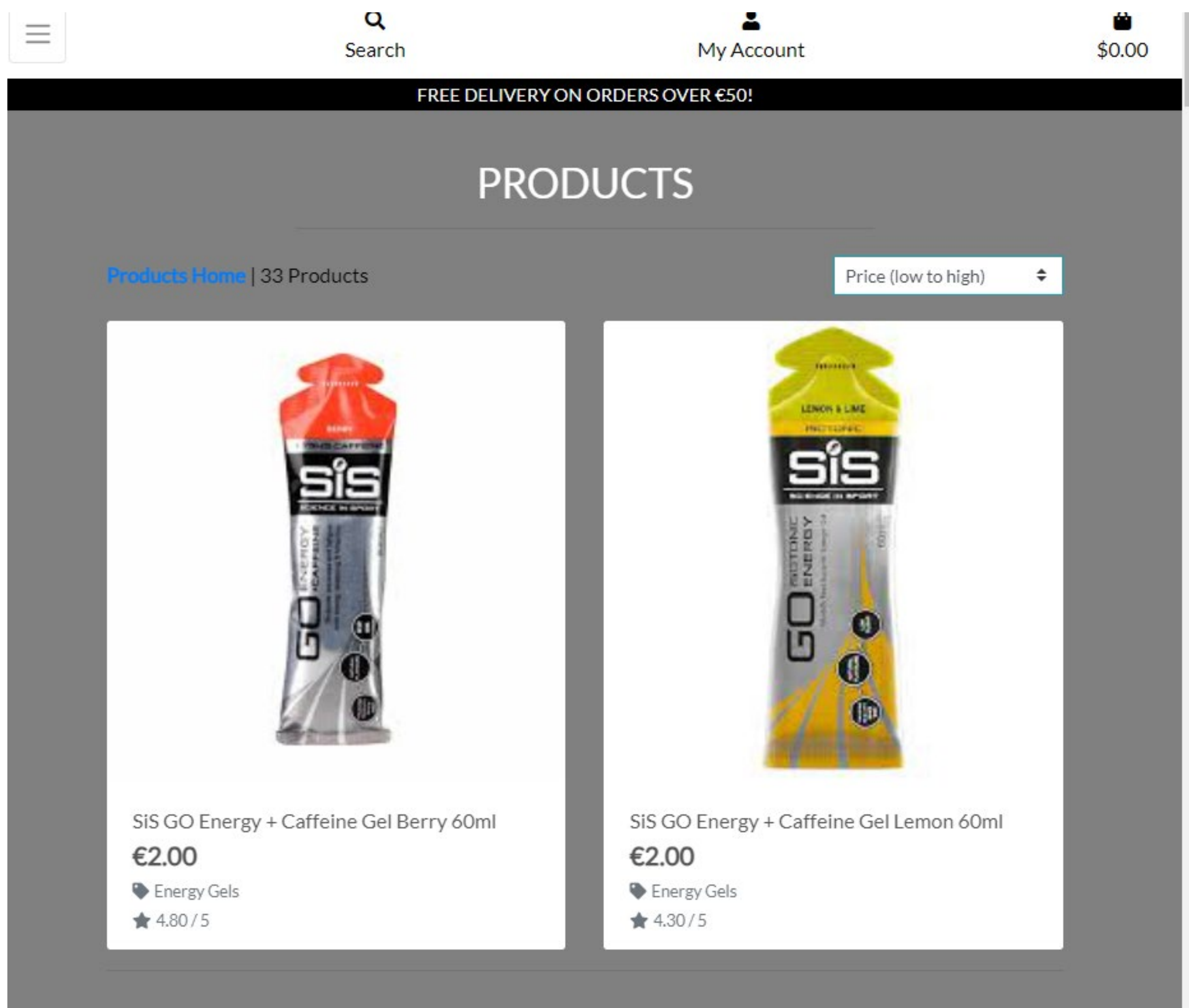
Homepage working on smaller screens:



Navbar working on smaller screens:



Products working on smaller screens and only displaying 2 products per column.



Working on smaller screens and only displaying one product per column.

Search

FREE DELIVERY ON ORDERS OVER €50!

PRODUCTS

Price (low to high)

Products Home | 33 Products

SiS GO Energy + Caffeine Gel Berry 60ml

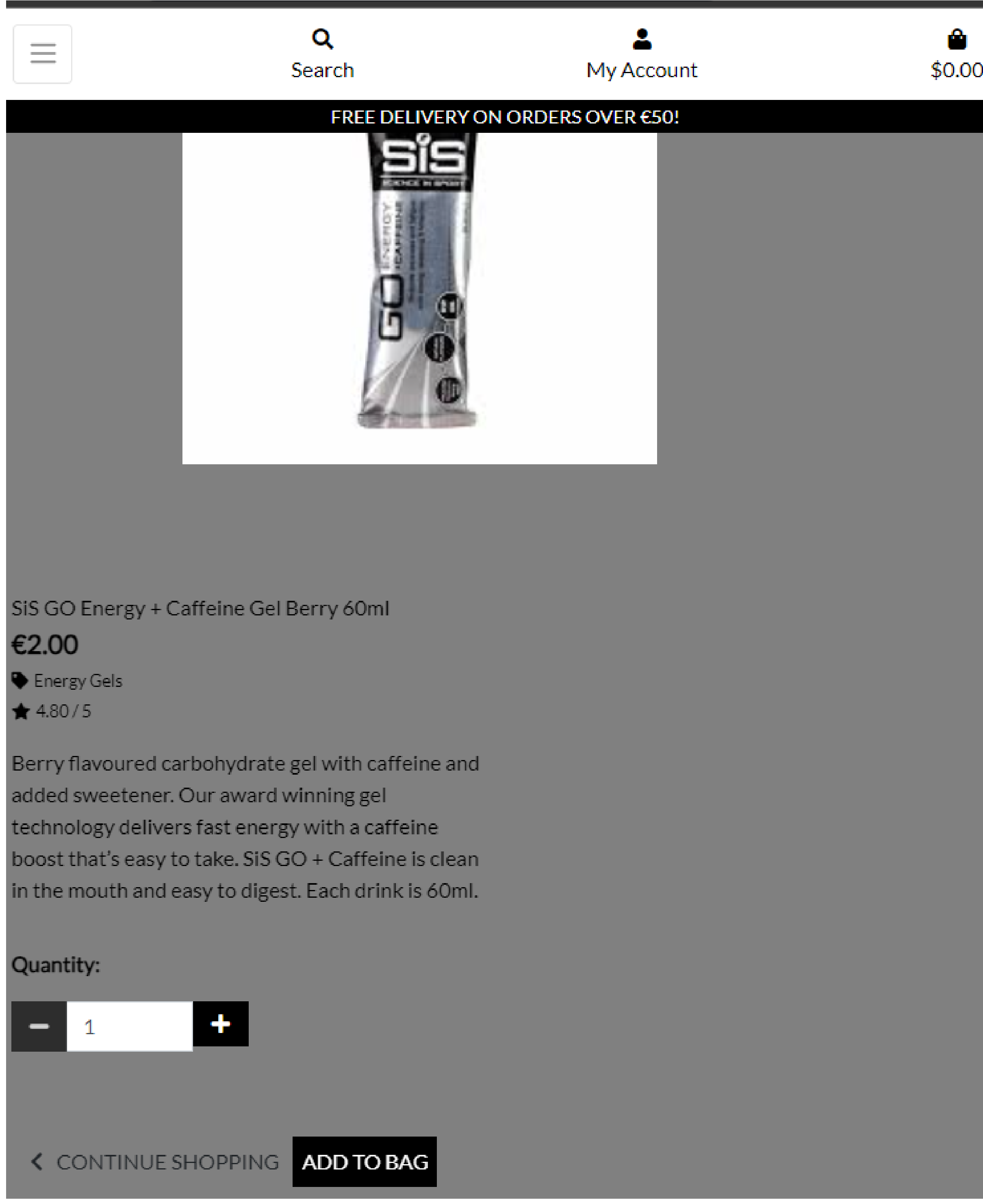
€2.00

Energy Gels

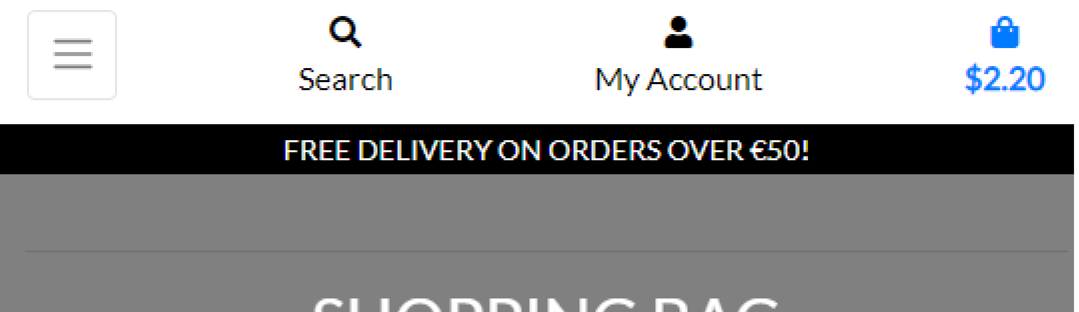
4.80 / 5

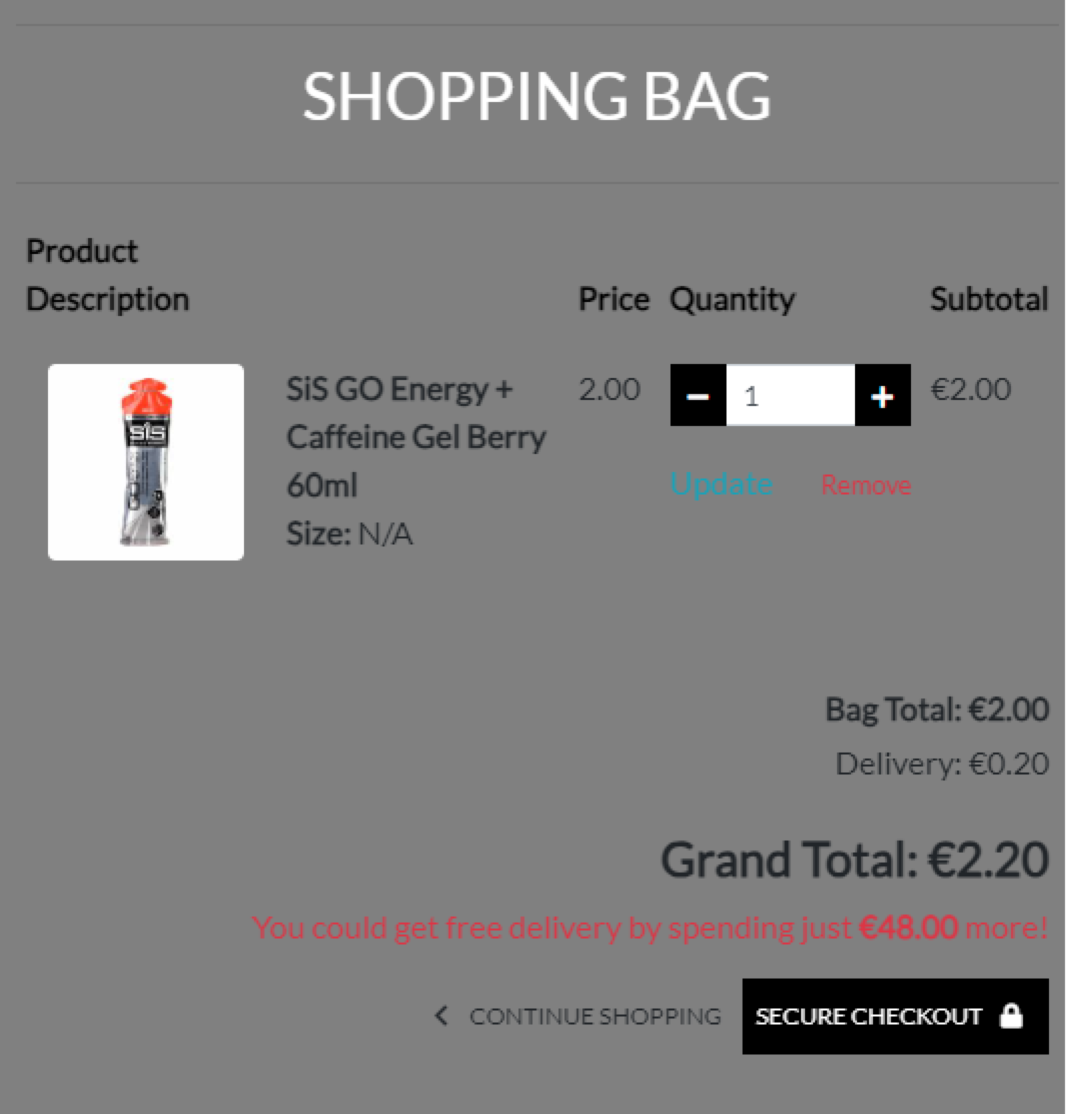
.gitpod.io/products/24

Product details working on smaller screens.



Shopping bag working on smaller screens.





Checkout working on smaller screens.

Search

My Account

$2.20

FREE DELIVERY ON ORDERS OVER €50!

CHECKOUT

Order Summary (1)

Item	Subtotal
SiS GO Energy + Caffeine Gel Berry 60ml Size: N/A Qty: 1	€2.00

Order Total:	€2.00
Delivery:	€0.20
Grand Total:	**€ 2.20**

Please fill out the form below to complete your order

Details

Colin Shaw

colin.shaw400@outlook.com

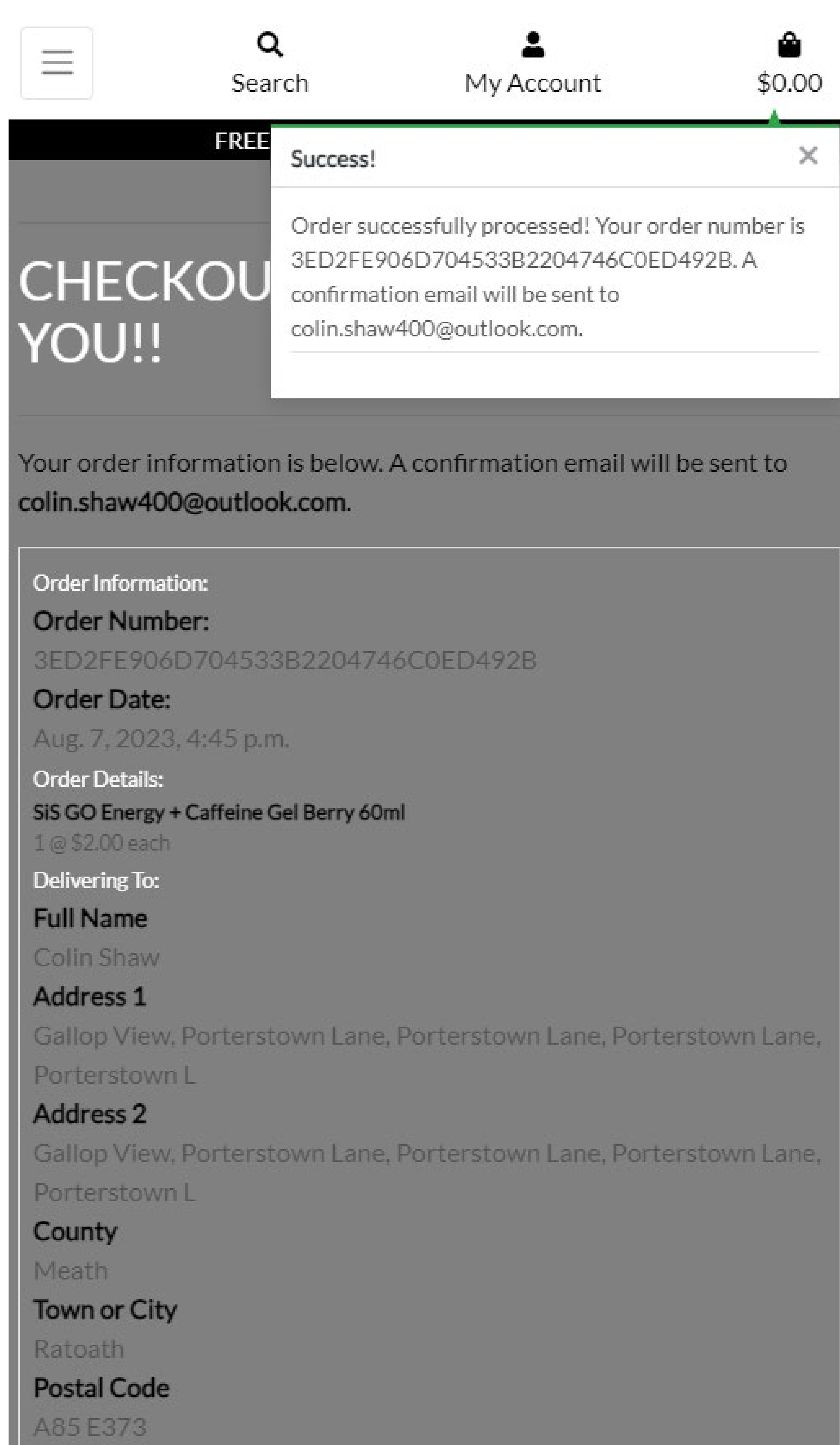
Delivery

+353851252606

Gallop View, Porterstown Lane, Porterstown Lane, Porterstown

Porterstown Lane

Checkout success working on smaller screens.


Search
My Account
$0.00
FREE
Success!
Order successfully processed! Your order number is 3ED2FE906D704533B2204746C0ED492B. A confirmation email will be sent to colin.shaw400@outlook.com.
CHECKOU
YOU!!
Your order information is below. A confirmation email will be sent to colin.shaw400@outlook.com.
Order Information:
Order Number:
3ED2FE906D704533B2204746C0ED492B
Order Date:
Aug. 7, 2023, 4:45 p.m.
Order Details:
SiS GO Energy + Caffeine Gel Berry 60ml
1 @ $2.00 each
Delivering To:
Full Name
Colin Shaw
Address 1
Gallop View, Porterstown Lane, Porterstown Lane, Porterstown Lane, Porterstown L
Address 2
Gallop View, Porterstown Lane, Porterstown Lane, Porterstown Lane, Porterstown L
County
Meath
Town or City
Ratoath
Postal Code
A85 E373
Country

My profile page working on smaller screens.

FREE DELIVERY ON ORDERS OVER €50!

PROFILE

Default Delivery Information

+353851252606

Gallop View, Porterstown Lane, Porterstown Lane, Porterstown La

Porterstown Lane

Ratoath

Meath

A85 E373

Ireland

Order History

UPDATE INFORMATION

Order Number	Date	Items	Order Total
B65AC...	Aug. 7, 2023, 4:36 p.m.	SiS GO Energy + Caffeine Gel Berry 60ml x1	$2.20
92F19...	Aug. 7, 2023, 4:40 p.m.	SiS GO Energy + Caffeine Gel Berry 60ml x1	$2.20

Allowed hosts:

Adding the environment to allow hosts.

```
DisallowedHost at /
Invalid HTTP_HOST header: '8000-colinshaw1-starnutritio-idqu9tneld4.ws-eu100.gitpod.io'. You may need to add '8000-colinshaw1-starnutritio-idqu9tneld4.ws-eu100.gitpod.io' to ALLOWED_HOSTS.
Request Method: GET
Request URL: http://8000-colinshaw1-starnutritio-idqu9tneld4.ws-eu100.gitpod.io/
Django Version: 3.2
Exception Type: DisallowedHost
Exception Value: Invalid HTTP_HOST header: '8000-colinshaw1-starnutritio-idqu9tneld4.ws-eu100.gitpod.io'. You may need to add '8000-colinshaw1-starnutritio-idqu9tneld4.ws-eu100.gitpod.io' to ALLOWED_HOSTS.
Exception Location: /workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages/django/http/request.py, line 149, in get_host
Python Executable: /home/gitpod/.pyenv/versions/3.11.1/bin/python3
Python Version: 3.11.1
Python Path: ['/workspace/star-nutrition',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python311.zip',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python3.11',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python3.11/lib-dynload',
 '/workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python3.11/site-packages']
Server time: Sun, 25 Jun 2023 10:33:34 +0000

Traceback Switch to copy-and-paste view
/workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages/django/core/handlers/exception.py, line 47, in inner
 47.            response = get_response(request)
 ▶ Local vars
/workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages/django/utils/deprecation.py, line 116, in __call__
 116.         response = self.process_request(request)
 ▶ Local vars
/workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages/django/middleware/common.py, line 48, in process_request
 48.      host = request.get_host()
 ▶ Local vars
/workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages/django/http/request.py, line 149, in get_host
 149.         raise DisallowedHost(msg)
 ▶ Local vars

Request information
USER    [unable to retrieve the current user]
GET     No GET data
```

Fix add the below to settings.py files allowed hosts.

```
6   # SECURITY WARNING: don't run with debug turned on in production!
7   DEBUG = True
8
9   ALLOWED_HOSTS = ['8000-colinshaw1-starnutritio-idqu9tneld4.ws-eu100.gitpod.io']
0
1
2   # Application definition
```

Installing a pillow to stop pixilated images.

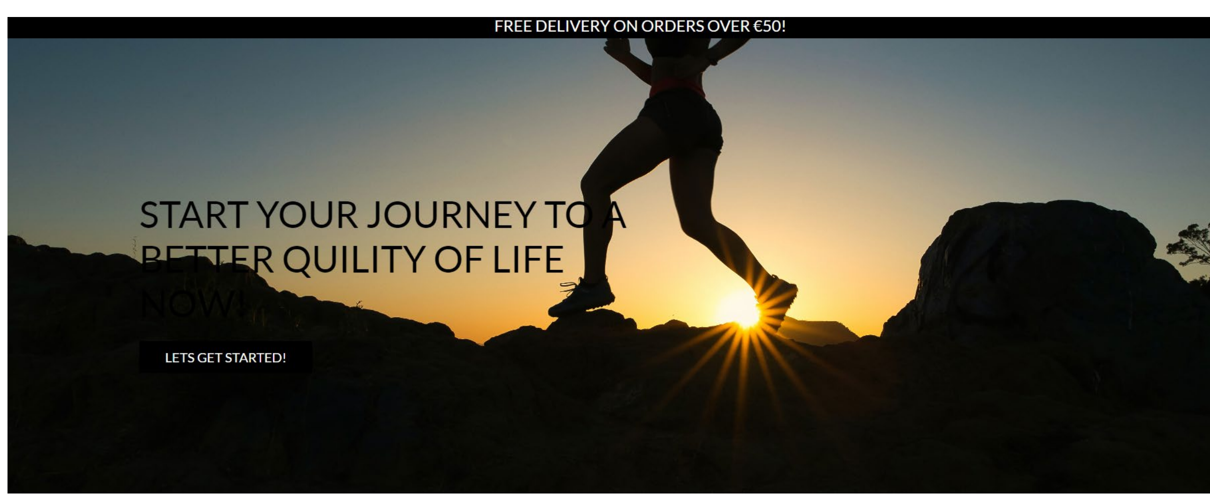
```
Not Found: /favicon.ico
[25/Jun/2023 11:56:55] "GET /favicon.ico HTTP/1.1" 404 2332
^Cgitpod /workspace/star-nutrition (main) $ pip3 install pillow
Collecting pillow
  Downloading Pillow-9.5.0-cp311-cp311-manylinux_2_28_x86_64.whl (3.4 MB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 3.4/3.4 MB 28.0 MB/s eta 0:00:00
Installing collected packages: pillow
Successfully installed pillow-9.5.0

[notice] A new release of pip is available: 23.0.1 -> 23.1.2
[notice] To update, run: pip install --upgrade pip
gitpod /workspace/star-nutrition (main) $
```

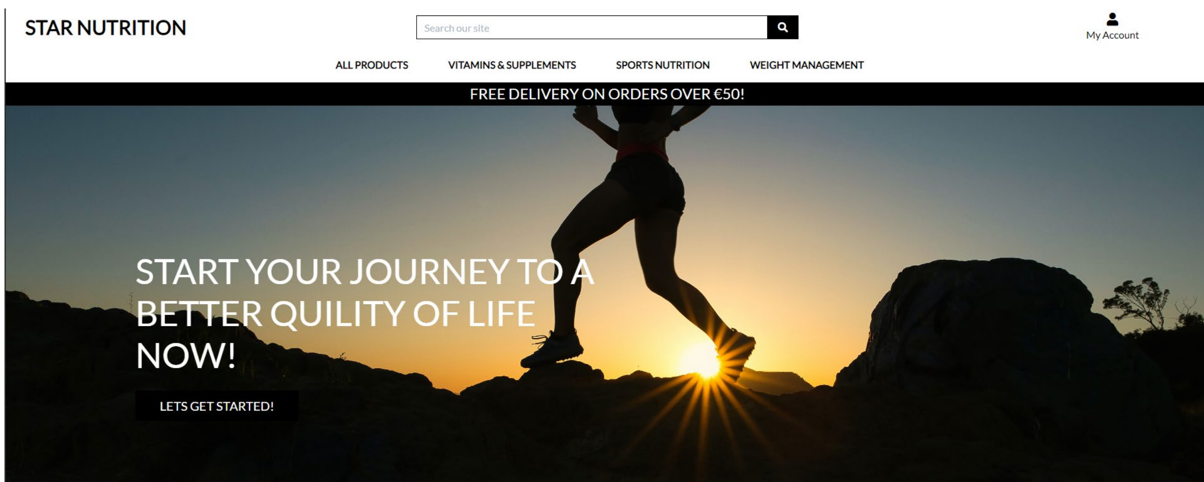
Had to install an older version of crispy forms to authenticate checkout payments.

```
AuthenticationError at /checkout/
You did not provide an API key. You need to provide your API key in the Authorization header, using Bearer auth (e.g. 'Authorization: Bearer YOUR_SECRET_KEY'). See https://stripe.com/docs/api#authentication for details, or we can help at https://support.stripe.com/.
Request Method: GET
Request URL: http://8000-colinshaw1-starnutritio-w7qx64xn9tn.ws-eu102.gitpod.io/checkout/
Django Version: 4.2.3
Exception Type: AuthenticationError
Exception Value: You did not provide an API key. You need to provide your API key in the Authorization header, using Bearer auth (e.g. 'Authorization: Bearer YOUR_SECRET_KEY'). See https://stripe.com/docs/api#authentication for details, or we can help at https://support.stripe.com/.
Exception Location: /workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages/stripe/api_requestor.py, line 159, in handle_error_response
Raised during: checkout.views.checkout
Python Executable: /home/gitpod/.pyenv/versions/3.11.1/bin/python3
Python Version: 3.11.1
Python Path: ['/workspace/star-nutrition',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python311.zip',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python3.11',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python3.11/lib-dynload',
 '/workspace/.pyenv_mirror/user/current/lib/python3.11/site-packages',
 '/home/gitpod/.pyenv/versions/3.11.1/lib/python3.11/site-packages']
```

```
[notice] To update, run: pip install --upgrade pip
gitpod /workspace/star-nutrition (main) $ pip3 install django-crispy-forms==1.14.0
Collecting django-crispy-forms==1.14.0
  Downloading django_crispy_forms-1.14.0-py3-none-any.whl (133 kB)
     ━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━━ 133.3/133.3 kB 4.4 MB/s eta 0:00:00
Installing collected packages: django-crispy-forms
```

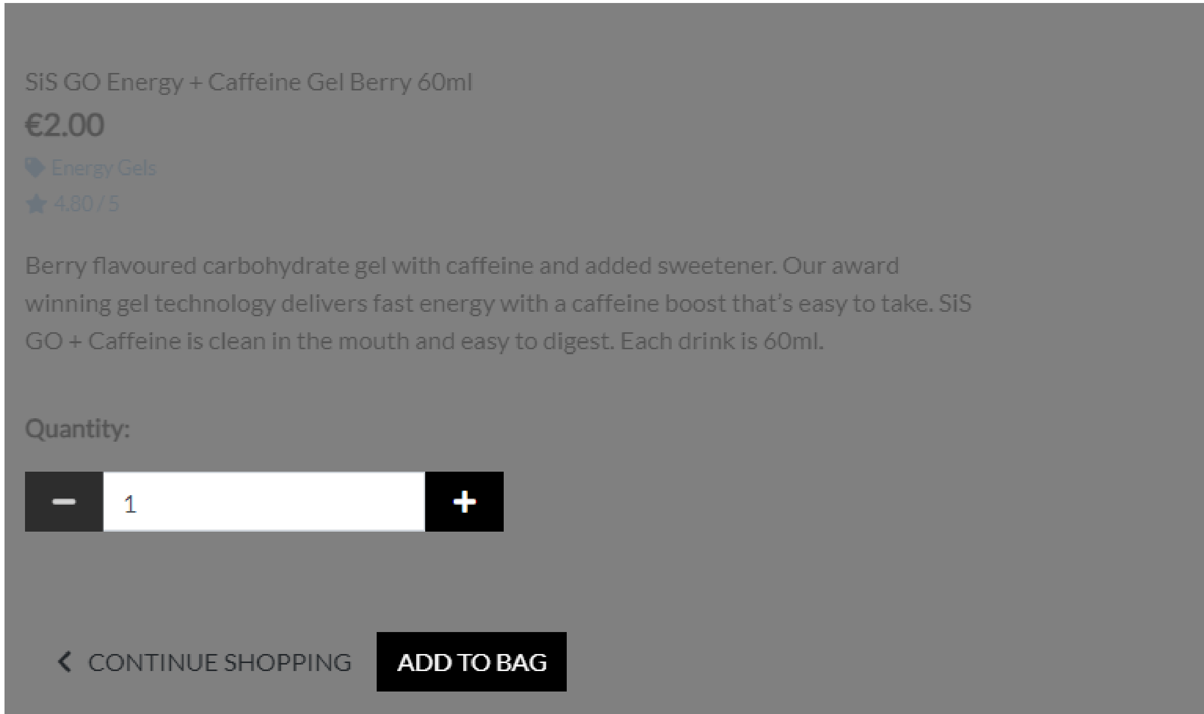
The homepage image has been pixilated had to install pillow and change the image.



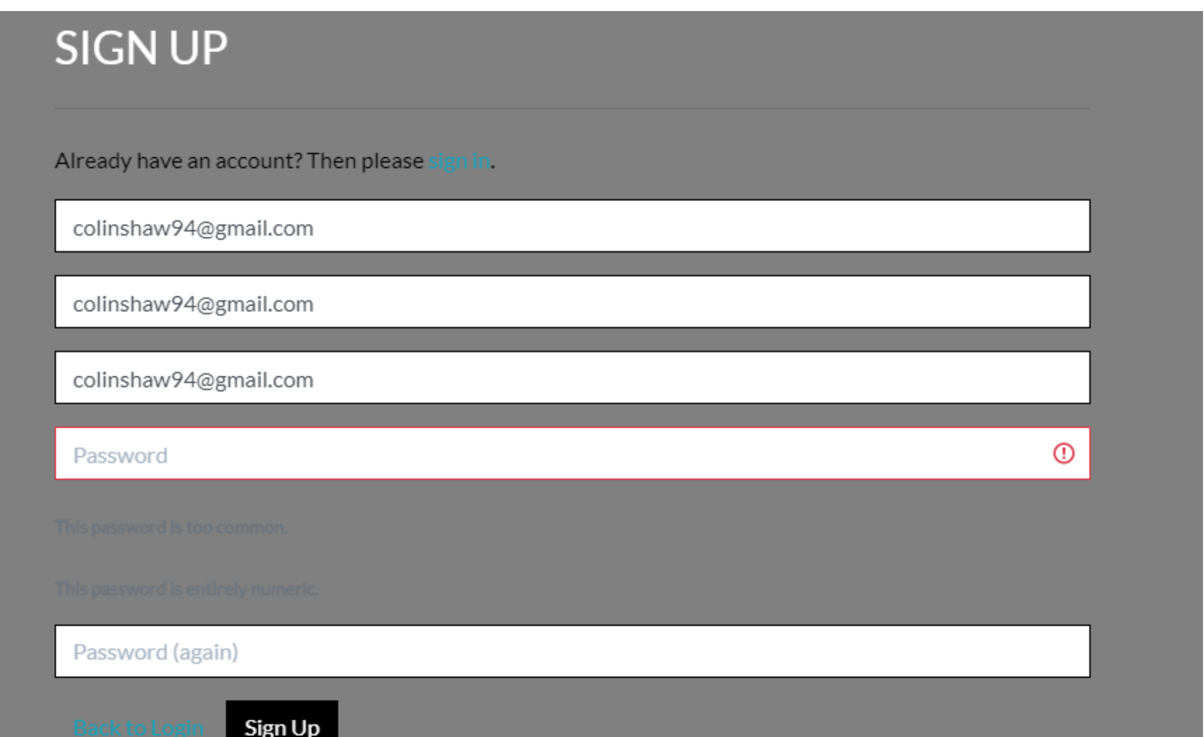


After testing on multiple devices the text was not good visually had to change the colour for certain screens.

Before



After



FREE DELIVERY ON ORDERS OVER €50!

PASSWORD RESET

Forgotten your password? Enter your e-mail address below, and we'll send you an e-mail allowing you to reset it.

E-mail address

Back to Login Reset My Password

Please contact us if you have any trouble resetting your password.

2.6. Evaluation

Overall the system is quick and responsive and dose everything it is asked to do with no errors or bugs.

The system was evaluated on mobile devices, laptop screens, iPad screens and larger screens via the developer tools on google chrome.

The code passed validation tests and gives no errors on the virtual studio code.

The system is well able for scalability and it will be very easy to add products and more growth in the future.

Correctness: Extensive testing and bug fixing ensured the system's correctness and robustness. This is evident in the above testing section and the use case approach to building the application ensured continuous testing while building.

3.0 Conclusions

Advantages:

Django helped in a rapid development environment.

User-friendly interface with a seamless shopping experience.

Efficient search algorithms enhance user satisfaction.

The scalable and well-organized architecture enables future expansions.

Search Engine Optimisation was easy with Djangos URL structure to help SEO performance.

Security was handled by Django with its built-in protection against SQL injection and cross-site scripting, and cross-site request forgery.

Disadvantages:

Lack of support for multiple languages.

Lack of products.

No clothes or accessories.

Not deployed to the cloud.

4.0 Further Development or Research

Deployment:

Due to the deadline being brought forward three weeks further than we had expected to submit our document and code, some things had to give to apply on time. The application would have been deployed through Heroku and AWS for static files.

Database:

the database would have been transferred to an Elephant SQL database for deployment to keep it secure when deployed on the cloud.

Marketing:

More marketing campaigns and revenue would lead to better products and more ideas around the application.

Social media advertainment:

Social media adds more revenue.

Reviews:

In future versions of the applications, customers will be able to write and post their reviews on products.

Likes:

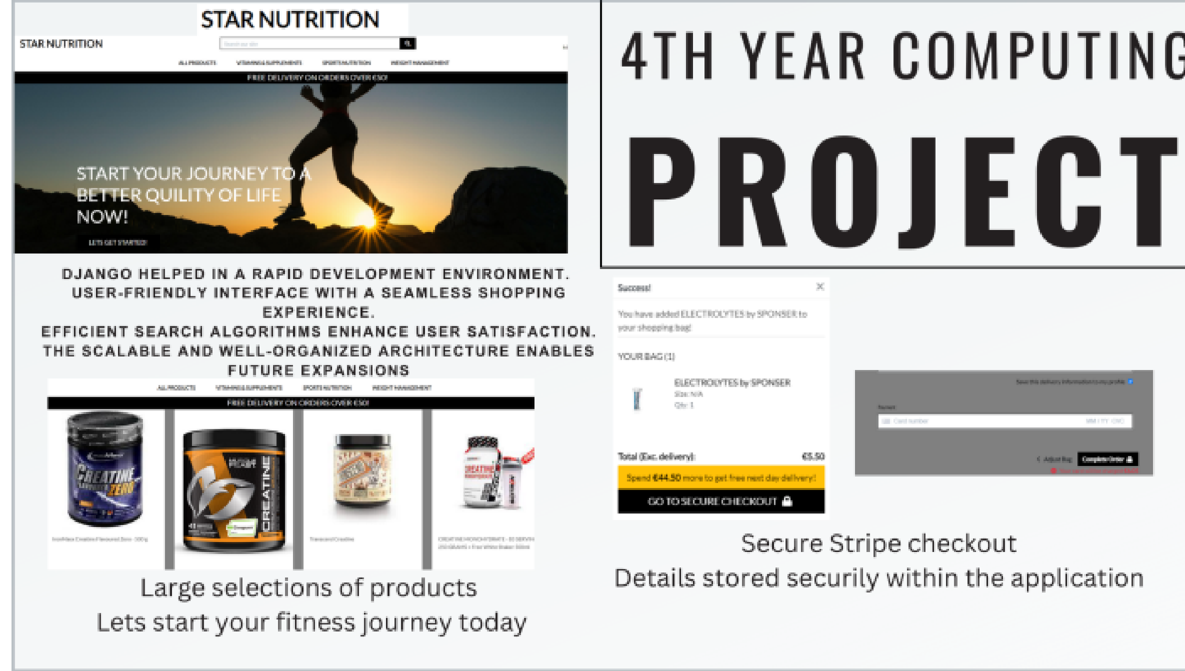
Customers will be able to like the products they have bought and be able to build an algorithm to show products that they might like.

Subscriptions:

Customers will be able to subscribe to certain products for delivery every month and update the subscriptions.

5.0Appendices

4.1. Poster



4.2. Project Proposal

Proposal Template Software Cyber DBT