Table of Contents

Executive Summary

1. Project Overview 4

1.1. Background 4

1.2. Aims 5

1.3. Technologies 6

1.3.1. Front-end 6

1.3.2. Back-end 6

1.3.3. Hosting services 8

1.4. Structure 8

2. System 9

2.1. Requirements 9

2.1.1. Functional requirements 9

2.1.1.1. Top level use-case diagram 10

2.1.2. Data requirements 11

2.1.3. User requirements 11

2.1.4. Usability requirements 12

2.2. Design and Architecture 12

2.2.1. Application Architecture 12

2.2.2. Database Design. 13

2.3. Project Environment and Directory Structure. 15

2.4. Implementation. 17

2.4.1. Annotations. 17

2.4.2. View Product Component. 18

2.4.2.1. View Product Component Code. 20

2.4.3. Register Component 28

2.4.3.1. Register Component Code 29

2.4.4. Login Component. 35

2.4.4.1. Login Component code. 36

2.4.5. Add to Cart Component. 43

2.4.5.1. Add to Cart Component code. 45

2.4.6. Upload Prescription component. 49

2.4.6.1. Upload Prescription component code. 51

2.4.7. Make Payment component. 59

2.4.7.1. Make Payment component code. 60

2.4.8. Add Product Component. 66

2.4.8.1. Add Product Component code 67

2.4.9. Extra Components coded. 73

2.5. Testing 74

2.5.1. View Product Component. 74

2.5.2. Register Component. 76

2.5.3. Login Component 78

2.5.4. Add to Cart Component. 79

2.5.5. Upload Prescription Component. 81

2.5.6. Add Payment Component. 83

2.5.7. Add product Component. 84

2.6. Graphical User Interface (GUI) Layout. 85

2.6.1. Homepage 89

2.6.2. Login. 92

2.6.3. Register. 93

2.6.4. Shop 94

2.6.5. Cart. 96

2.6.6. Checkout. 97

2.6.7. Payment. 99

2.6.8. Profile. 99

2.6.9. CMS Homepage. 102

2.6.10. CMS Category and Product. 103

2.7. Customer Testing 106

2.8. Evaluation. 108

2.8.1 Google Dev Tools - LightHouse. 108

3. Conclusions. 109

4. Further development or research. 111

5. References 113

6. Appendix: 114

6.1. Credentials. 114

6.2. Specification Document. 115

6.3. Product Design Document 138

6.4. Project Plan 161

6.5. Acronyms, Abbreviations and Terms 162

6.6. Other Materials Used. 164

Executive Summary

Specsavers are one of the largest retailers of prescription glasses in Ireland and recently they have increased their annual profits from €16.7m to €18.3m dating from February 2021 of the previous financial year(*Taylor C, 2021*) Specsavers offer a wide range of products and services. They provide the whole spectrum of eye care from eye tests, diagnosing a prescription, to selling and servicing glasses and contact lenses. If consumers wish to purchase basic set of frames with prescription lenses from Specsavers the price starts from €25 at the lowest end(*Specsavers, 2021*). Similarly another large retailer VisionExpress, who have 559 stores between the U.K and Ireland(*VisionExpress, 2021*), offer a similar value proposition with their lowest end prescription glasses from €25-€30.

SimpleFocals, a new online prescription glasses retailer aims to disrupt these large vendors at the lower-segment of the prescription eyewear market. SimpleFocals aims to give the Irish prescription wearing consumer a choice of different style frames at a price point between €10-€25. The scope of this project is to develop an e-commerce store that facilities an online customer all the functions nessecary in order to purchase a pair of prescription glasses by interacting with the web application.

The application development was accomplished using a combination of front-end and backend technologies in order to achieve a working prototype. The application was tested thoroughly throughout the development process in order to see if the functional parameters of the application were satisfied. The planning, design and implementation of this project has resulted in the a web application that in future revisions will be ready for commercial deployment.

1. Introduction

The introduction section describes the background of the SimpleFocals concept, the aims of this project, the technologies used within this project and the overall format of this technical document.

1.1. Background

I have always had an interested in E-commerce and having studied it to masters level at university, it was always been my intention to create a project in the space. The inception of this

project was derived from my experience of living in Asia for many years. I was always impressed with the quality of eye wear products relative to their cost. I have owned many prescription glasses that were purchased from Asia and I was fully satisfied with them. In research for SimpleFocals I connected with suppliers in China, specifically on aliexpress that will supply stock for the store. The unit price for each item can be purchased from a cost of between €3-€10 for frames and customised prescription for each lens. The selling price of each item from €10-€25 euro offers a good mark-up stock purchase price and compares favourably when compared to large retailers Specsavers and VisionExpress in their lower end-tier offerings at €25-€30. I believe there is still room for more competitors in this space as revenue for the eyewear market in Ireland during 2021 was €360m and between the 2021-2025 the market is projected to increase by 4.45% annually (*Statista, 2021*). Indications suggest there is still opportunity in the eyewear space as the market is increasing at a steady rate each year.

In research for this project I discovered the sale of eyewear in Ireland has a stipulation. Irish legislation for the sale and distribution of spectacles (*Iris Oifigiúil, 2015*) Article8 states: "*No **dispensing optician** or optometrist shall sell or offer for sale any spectacles **except on a prescription issued** following an examination by a registered medical practitioner or by an optometrist*". This requirement shall be developed in the application. If a customer wishes to purchase prescription eyewear from SimpleFocals they are required to upload a valid prescription that was received from an optometrist.

1.2. Aims

The aim of this project is to create a functioning e-commerce store for the business SimpleFocals. The web application is the entry point that will allow consumers to easily browse SimpleFocals range of prescription eyewear, easily order and pay for glasses. The web application will be hosted online and accessible and usable on multiple device from factors from smartphone, tablet and computer. The specific target market for SimpleFocals are people who wear (myopic) prescription glasses but want a cheaper alternative to most other eyewear retailers (online or physical stores). At the conclusion of this project, the developed application should form the basis for a commercially deployable e-commerce store(given future revisions).

1.3. Technologies

The web application was developed using a combination of front-end and back-end technologies in order to achieve a working prototype for the e-commerce store. These technologies are also described in the ***Product Specification Document.***

1.3.1. Front-End

These front-end technologies and resources were used in the development of the web application.

Bootstrap4.0: Bootstrap is a CSS framework that was used to develop the layout webpages for the application. It has the benefit of being mobile responsive and it aided in quick development of the website for other form factors.

HTML5: HyperText Markup Language HTML is the standard language for displaying information on a web browser. Bootstrap classes were used in combination with HTML tags to define the layout of the e-commerce store.

CSS: Cascading Style Sheet CSS, was used for the presentation of design styles in the web application.

Javascript: Javascript is a language that is used to manipulate objects in the DOM.

JQuery: JQuery is a feature rich Javascript library that is used to manipulate the DOM. During development JQuery was used to when uploading a file/image to the system.

Font Awesome: Online resource for icons that are embedded in the webpages. Used for the facebook, instagram, twitter and linkedin during development.

1.3.2. Back-End

These back-end technologies and resources were used in the development of the web application.

Java11: Java is a high level object orientated programming language that was used in the development of the back-end of the application.

Spring boot: Spring boot is a subset of the Spring Framework that allows developers to concentrate on coding applications quickly without having to deal with complex project configurations. Spring boot is the ideal platform for developing for web application in Java. Dependancies can be added to the project as and when specific functionalities are needed.

MySQL DB: MySQL DB is free open-source relational database software that is developed by Oracle. This database was used as the persistent storage in the application.

Hibernate: JPA (Java Persistence API) is the standard approach for Object Relational Mapping(ORM), ORM is where Java objects are mapped to data in a relational database. Hibernate is one of the most common implementations of JPA.

Spring-Boot-Starter: Spring boot starter is a stable of most Spring boot applications, it helps reduce the time the developer uses for searching and adding dependancies. Spring-boot-starter groups the require dependancies together in a single package.

-Web: Spring-Boot-Starter-Web is a library that is used for building RESTful web applications using Spring MVC. Spring MCV applications use Tomcat as the default web container.

-Data-JPA: Spring-Boot-Starter-Data-JPA is a data access abstraction that is used to reduce boiler plate code when implementing data access layer for persistence stores. In the project hibernate was used in combination with JPA libraries for persistent storage in the application's database.

-Security: Spring-Boot-Starter-Security is an authentication and access-control and framework that has become the standard for securing Spring applications. These libraries were used for the login and registration components of the application.

-Thymeleaf: Spring-Boot-Starter-Thymeleaf includes the all the libraries for Thymeleaf for use with Spring applications. Thymeleaf is a HTML5/XHTML/XML server side java templating engine. It was used in the development process to embed objects/data from the backend to front-end views within HTML tags.

-Test: Spring-Boot-Starter-Test is a set of libraries that support test classes. This was used in combination with classes in Spring-Boot-Dev tools to test methods in the development of the application.

Spring-Boot-Devtools: Spring-Boot-DevTools provided development tools for reloading the project and debug options.

Stripe API: Stripe API is the application programming interface provided by trusted third party Stripe in order to provide secure payment processing.

GSON: GSON Google's JSON serialiser/deserialiser was used in place of the native Jackson serialiser/deserialiser package that is default to Spring Boot applications so that the Stripe API wouldn't have any conflicts.

1.3.3. Hosting Services

The hosting service that the app is deployed on is the free tier of Heroku cloud platform. Heroku makes configuration and deployment of prototype applications an easy experience that is the reason it is chosen as the cloud provider.

1.4. Structure

This document shall be structured beginning with the executive summary. The Executive summary provides and overview of the project as a whole, reasons for why the project is being developed, a brief description of the technical aspects and a summarised project evaluation. Section 1, the introduction of this document, outlines the background to the project, what the project is hoping to accomplish, the technologies used in development and the structure of the report. Section 2, describes detailed aspects of the system. The system requirements, design and architecture of the application, the implementation, the testing, the graphical user interface

layout and evaluations will be discussed. Section 3, outlines the conclusions that have been gathered from the developing the project. Section 4, describes how development and research could progress the project from its current form. Section 5 is the references used during the project development lifecycle. Section 6 is the Appendix for all the terms and supporting materials used in the development of the project.

2. System

This section of the document describes details of the system. The system requirements, design and architecture of the application, the implementation, the testing, the graphical user interface layout and evaluations will be discussed.

2.1. Requirements

The requirements section will briefly outline the functional, the data, the user and the usability requirements. For more in depth documentation on the requirements of the project please refer to the appended ***Requirements Specification Document***.

2.1.1. Functional Requirements

The functional requirements describes what the system must be able to accomplish. The requirements for the system are ranked in terms of priority (*1 = highest priority,*
7 = lowest priority).

1. **View Product**:	A customer is able to view a product from the online store.
2. **Register:**	A non-registered customer is able to register with the system.
3. **Login:**	A user(customer, administrator) is able to login to the system.
4. **Add to Cart:**	A registered customer is able add to product/s to a cart.
5. **Upload Prescription:**	Registered customer is able to upload a digital copy of their eye prescription.
6. **Make Payment:**	Registered customer's payment is able to be processed.
7. **Add Product:**	Administrator is able to add products.

The functional requirements outlined in the ***Requirements Specification Document(3.1 Functional Requirements)*** remain the same but activation and main flow in some cases have been altered slightly because of graphical user interface layout alterations and designs considerations. The use-case descriptions in the ***Requirements Specification Document(3.1 Functional Requirements)*** have been updated to reflect the changes.

2.1.1.1. Top Level Use Case Diagram

The use case diagram outlines the actors and how they interact with the system. SimpleFocals System is described below(*fig2.1.1.1*).

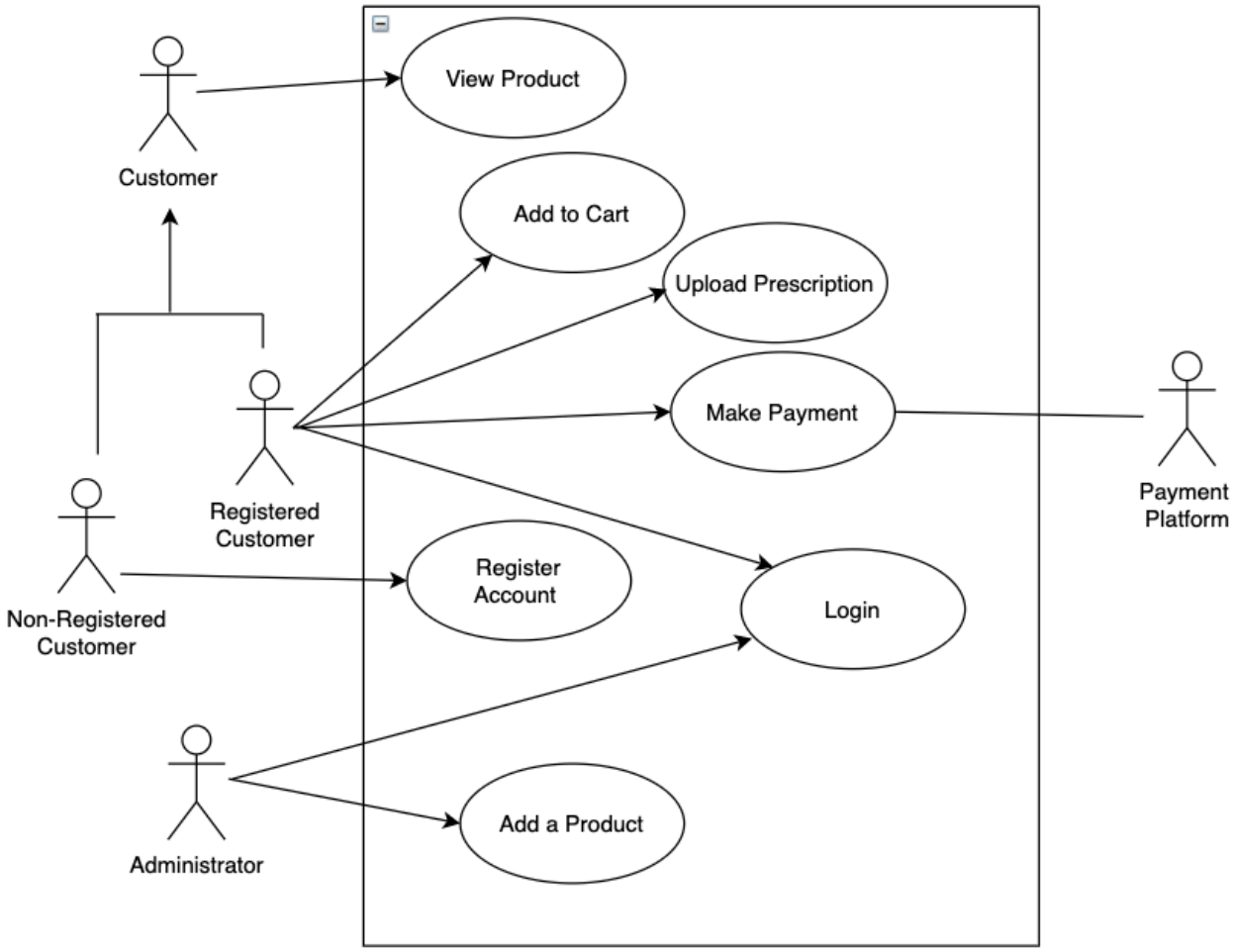

fig2.1.1.1: Top Level Use-Case Diagram for SimpleFocals System

2.1.2. Data Requirements

The data requirements describe how data in the system will be gathered and used. Data in the SimpleFocals system will be inputted by the customer and the administrator. The data inputted to the system will be saved to persistent storage and facilitate functions of the system.

- **Administrator:** The system will facilitate the administrator adding product/s to the system. The system will save details of each product. The name, the description, the price, the category and an image will be saved as attributes of the product. Once a product is saved to the system it will be viewable and accessible in the shop section of SimpleFocals e-commerce store.
- **Customer:** The system will facilitate a customer registering with the system. Registering with the system will save the attributes of first name, last name, email and password of the customer. When the customer logins into the system, they are required to fill out a customer profile which includes, the address, county, postcode, right and left eye prescription as well as a digital copy of their prescription. Once a customer profile has been entered in the system, the customer will have access to cart functionality. A customer will be able to add to cart. Cart information is persisted in the system. A customer is able to log in and out of the system and their cart information is retained by the system. When a customer proceeds to the payment page and enters details for payment, *the system will not save any sensitive card details(an external payment processor will securely handle the transaction)*, only a record of the amount payable for goods will be recorded by the system.

2.1.3. User Requirements

The user requirements outlines what a user needs in order to access and use the system. The user must have the following:

- **A Computing Device:** The computing device (computer, tablet or smartphone) should have hardware that supports connection to the internet.
- **An active internet connection:** The computing should have an active connection to the internet.
- **An internet browser:** The computing device should have a browser makes use of the internet connection in order to interface with the web application.

2.1.4. Usability Requirements

The usability requirements of the system specifies how easy the system is to navigate and use. When the user navigates to the URL of SimpleFocals web application. The user will be greeted by the home screen. From the home screen, the graphical user interface GUI is designed so that users who interact with the system will find it:

- **Efficient**: The functions which the user wishes to perform are responsive and as free from errors as possible.
- **Intuitive:** The functions the user wishes to perform can be gauged without using extra information or a manual in order to execute.
- **Easy to navigate:** The functions available to the user are clearly defined and easily accessibly.

2.2. Design and Architecture

This section describes the design and architecture of SimpleFocals web application. An overview of the application architecture and the database design shall be outlined. For more in depth documentation on the Design and Architecture of the system please refer to the appended ***Project Design Specification Document.***

2.2.1. Application Architecture

This section gives an overview of the application architecture of SimpleFocals web application. The application is developed as a spring boot application using a model view controller MVC architectural design pattern. The MVC design separates the application into three separate logical components, the model, the view and the controller. The model is the component that manages the data and the logic of the application. The view is the component that represents the information to the end user. The controller is the component that routes information to and from the model or the view. The image (*fig3.1.1*) shows the application architecture of the system. The view and the controller components are defined but the model is composed of service class, model class, repository interface and relational database storage.

The user's device browser is represented by client. The user accesses the web application by entering a URL corresponding to the SimpleFocals hosted website. When the user wishes to access a specific resource from the web application, it sends a request to the server, the appli-

cation's controller then processes the request. The controller communicates with the service layer to handle the application logic for the specified request. The application logic is the business rules that have been defined to implement application functionality, this can involve modifying data in persistent storage before returning a result back to the controller and the view. This communication flow is hidden from the user.

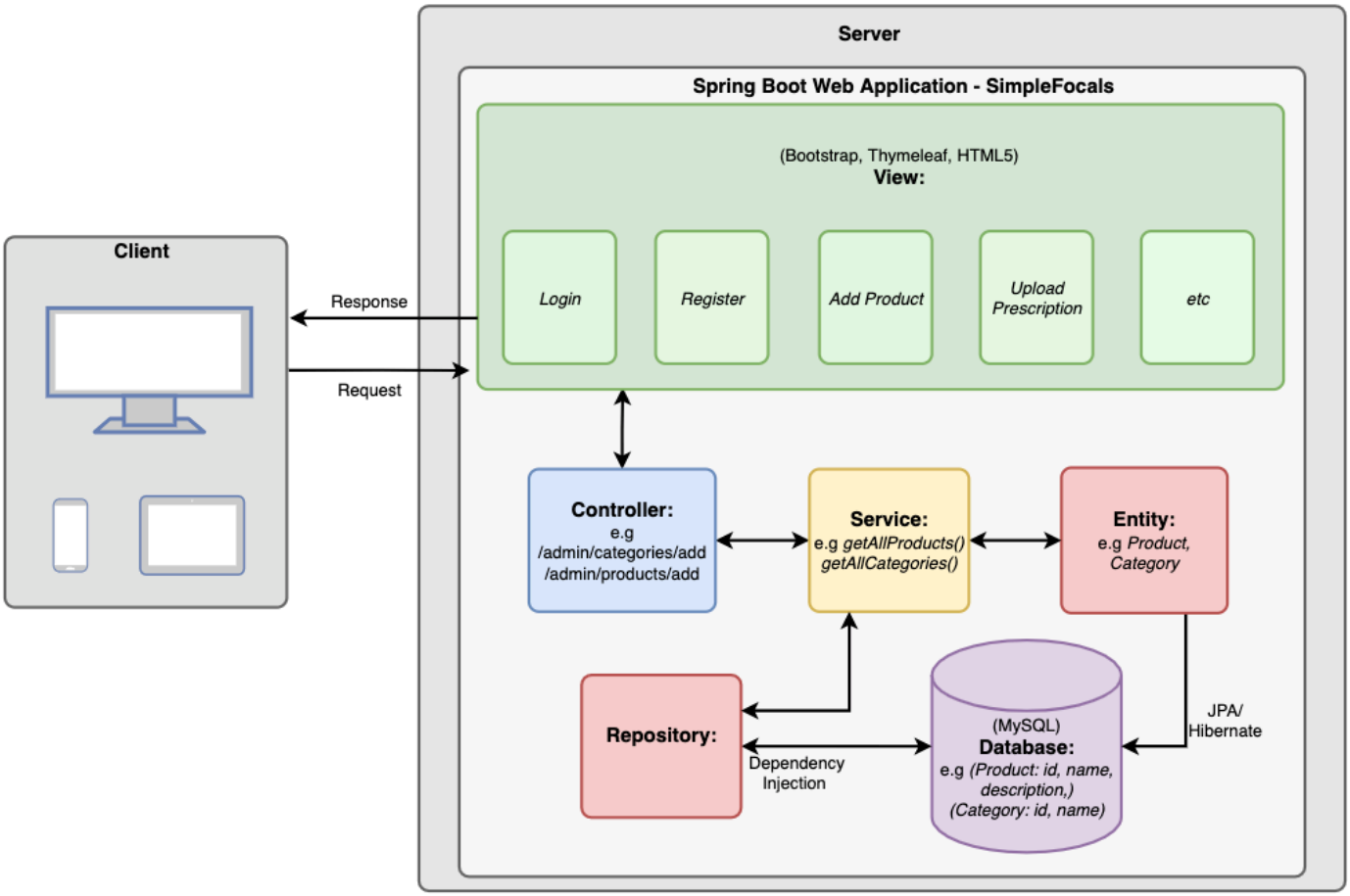

fig3.1.1: application architecture view of SimpleFocals

2.2.2. Database Design

This section gives a brief overview of the database system employed by the application and how it was designed. The conceptual design was the first step in the process for developing the relational database system. The core entities of the business were identified. Then the relationships and the cardinalities between the entities were defined. From this point an entity relation-

ship diagram of the system was created describing the overall database system(*fig2.2.3.1*). Lastly the tables were separated into normalised logical form.

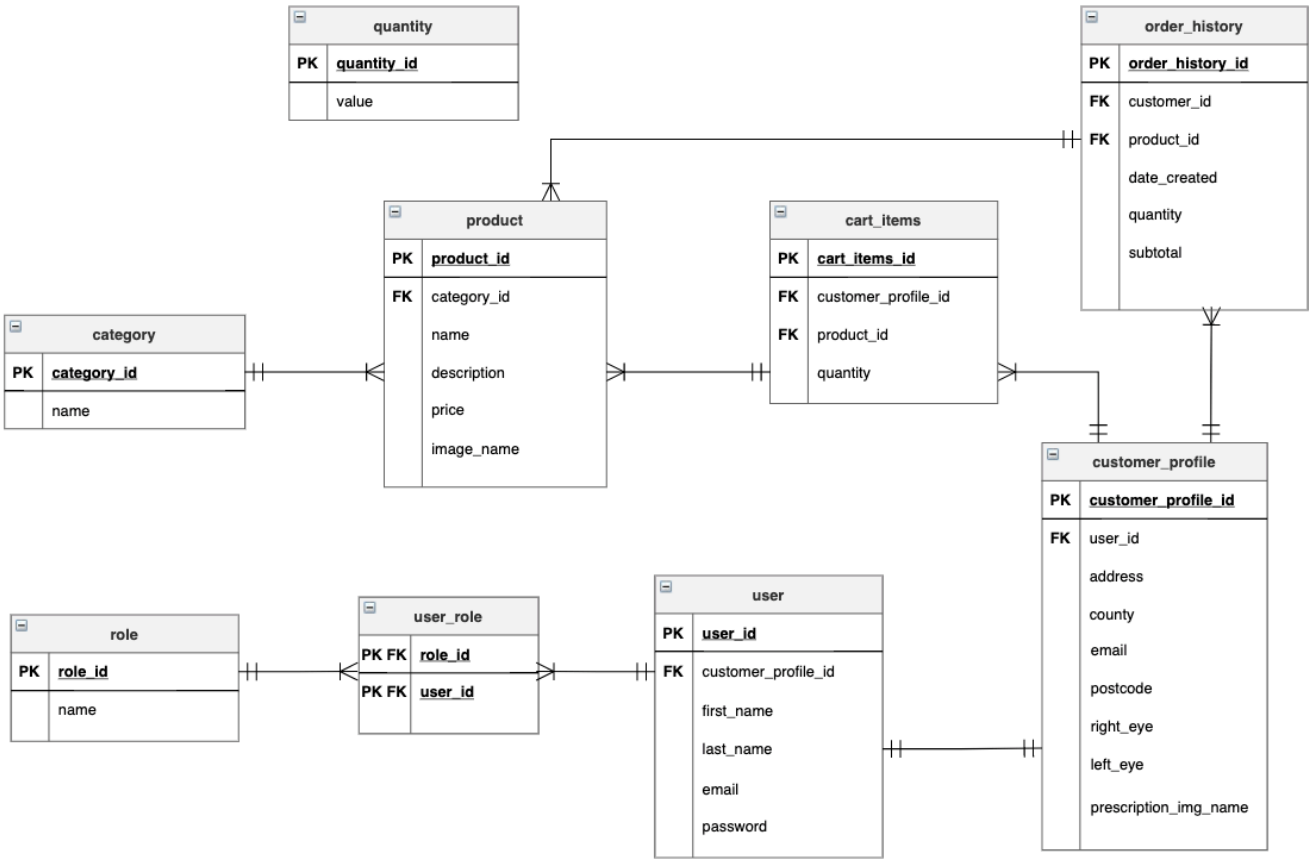

fig2.2.2.1: The ERD for SimpleFocals RDMS

An in depth description of the database design can be found in the appended ***Product Design Specification Document(4.2. Database Design).*** During the development of the application some changes were made to the initial database design plan. The updated changes are reflected in the appended ***Product Design Specification Document.***

2.3 Project Environment and Directory Structure

This project was developed using IntelliJ Ultimate Edition IDE. The benefits of developing in this environment were the ability to build projects using either maven or gradle. SimpleFocals project is a Maven build and the dependancies for the project were added in the pom.xml file in the project directory. IntelliJ also has the ability to install plug-ins, a plugin was installed to start the Spring Boot project build natively without the need for importing separate configuration

files. IntelliJ Ultimate Edition also has the ability to integrate a database management system so, changes to database elements can be monitored easily without the need to install any external software. A MySQL database is used in the project. This was implemented by setting configurations for the database in *src -> main -> resources: application.properties.* The datasource points to *:jdbc:mysql://localhost:3306/simplefocals1,* which the locally stored database.

The credentials for the database are: **SimpleFocals1:**

Username: root

Password: rootroot

The directory structure of the project is shown below. The project consists of many packages located in the *src -> main -> java -> com.github.simplefocals* path(*fig:2.3.1*). The main method for the application is SimlefocalsApplication.java that is also located in this directory.

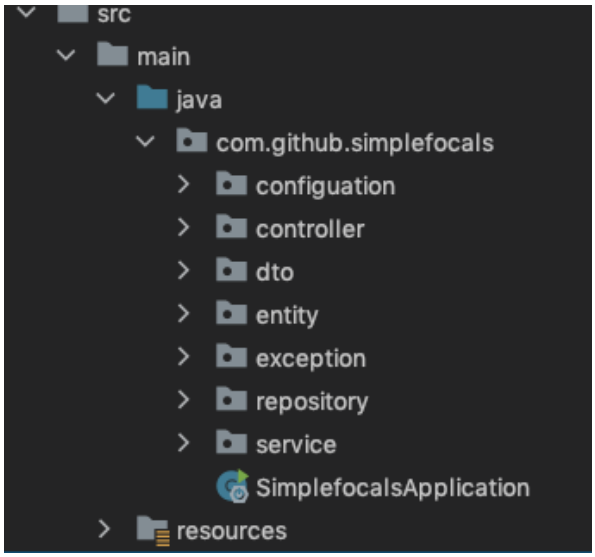

fig:2.3.1: Main packages

These packages include:

- **configuration:** security configuration files are here
- **controller:** the controller for Home, Login, Admin, Cart, Profile and Stripe Payment are here
- **dto:** CustomerDTO and ProductDTO files are here
- **exception:** custom exception files are here

- **entity:** Product, Category, User, Role, Customer Profile, Cart Items and Order History entity classes are here.
- **repository:** Cart Item, Category, Customer Profile, Order History, Product, Role and User repository files are here
- **service:** Category, Customer Profile, Custom User Details, Product, Role, Shopping Cart and User service files are here.

Resource directory stores all the static files and the templates required in the front-end of the application. Navigation to *src -> main -> resources -> static,* hosts all packages that contain static files needed for the application for run(*fig:2.3.2*). These packages include:

- **css:** contains all the css styles that are to be applied to the webpages
- **images:** contains all the images that are statically presented on the website
- **js:** contains the javascript files that are applied to the webpages when loaded
- **prescriptionImages:** This directory contains the prescription files that are uploaded from customers when they input their customer details.
- **productImages:** This directory contains the product images that an administrator can upload to the application when they use the add a product function in the CMS.

Navigation to *src -> main -> resources -> templates,* hosts all the html pages of the web application(*fig:2.3.3*).

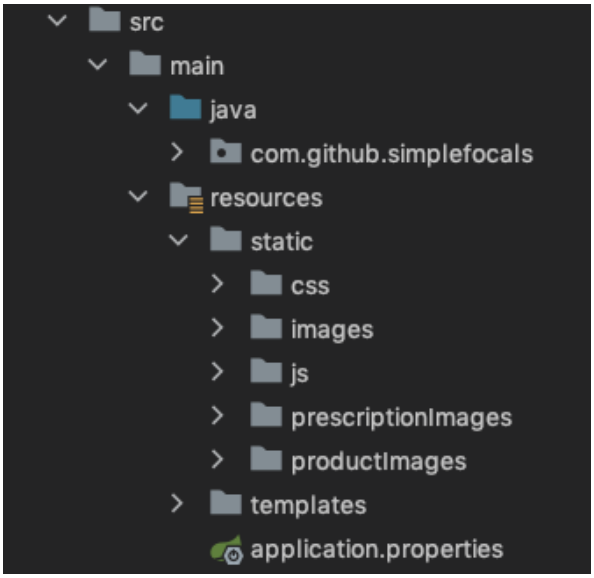

fig:2.3.2: resources static files

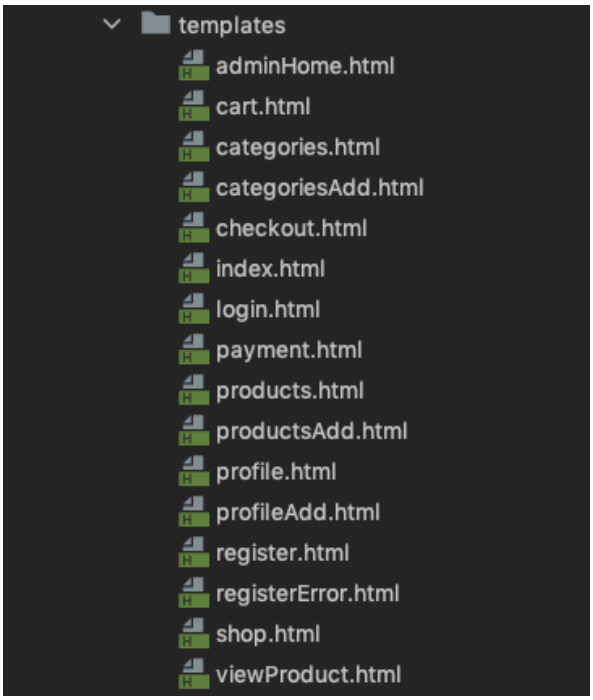

fig:2.3.3: html files in templates

2.4. Implementation

This section describes the implementation of the application. The functional requirements that are outlined in **2.1.1. Functional Requirements** of this document shall each be examined.

2.4.1. Annotations

Spring Boot facilitates the use of annotations. The annotations allow Spring applications to configure dependences and also implement dependancy injections in the application. Use of annotations helps to reduce a lot of boilerplate code that is required to perform essential functions. Types of annotations used in the implementation of the project:

- **@Table**: Allows specifying attributes about the table. e.g name a table
- **@Entity:** Defines this class is going to be mapped to a table in the database.
- **@Id:** Defines that a specific attribute will be the primary key in a table.
- **@GeneratedValue:** Defines the type incremental behaviour that a primary key column has when a new entry is added to the table.
- **@ManyToMany:** Defines a many to many relationship between two tables.
- **@ManyToOne:** Defines a many to one relationship between two tables.
- **@JoinColumn:** Defines at which attribute a table should join at.
- **@Column:** Allows specifying attributes about the column. e.g name a column.
- **@Repository:** Is a specialisation of the @Componet annotation, repository annotation is used as a mechanisms for encapsulating and accessing data from persistent storage.
- **@Service:** Is a specialisation of the @Componet annotation, service annotation is used in order to provide business logic functionalities for a specific class.
- **@Controller:** Is a specialisation of the @Componet annotation, controller annotation indicates this class acts as the role of controller in the Spring MVC framework.
- **@ModelAttribute:** Is used to inject an objects in the model. This is used in-conjunction with Thymeleaf to inject data objects to the front-end views. e.g a User object.
- **@RequestParam:** Is used to inject single parameter of data. e.g firstName attribute.
- **@PathVariable:** Used to pass values in the URI. e.g sets the value for id = 2 for the resource at /shop/viewProduct/{id}

- **@Autowired:** Is used to inject object dependancy where functionality is needed. e.g autowire a ProductRepository object in the ProductService class.
- **@GetMapping:** Is used for mapping HTTP GET requests for specific handlers.
- **@PostMapping:** Is used to mapping HTTP POST requests for specific handlers.

2.4.2. View Product Component

This component describes how non-registered and registered customers of the system are able to view items in the shop section of the web application. A customer is able to view products via shop, filter by category and click into a more detailed view of a product. The image(*fig2.4.2.1*) shows the class diagram of the view product function of the system.

The HomeController is the class in the application that routes requests for the shop resources that a user of the system wishes to access. From image(*fig2.4.2.1*), the resources accessible for

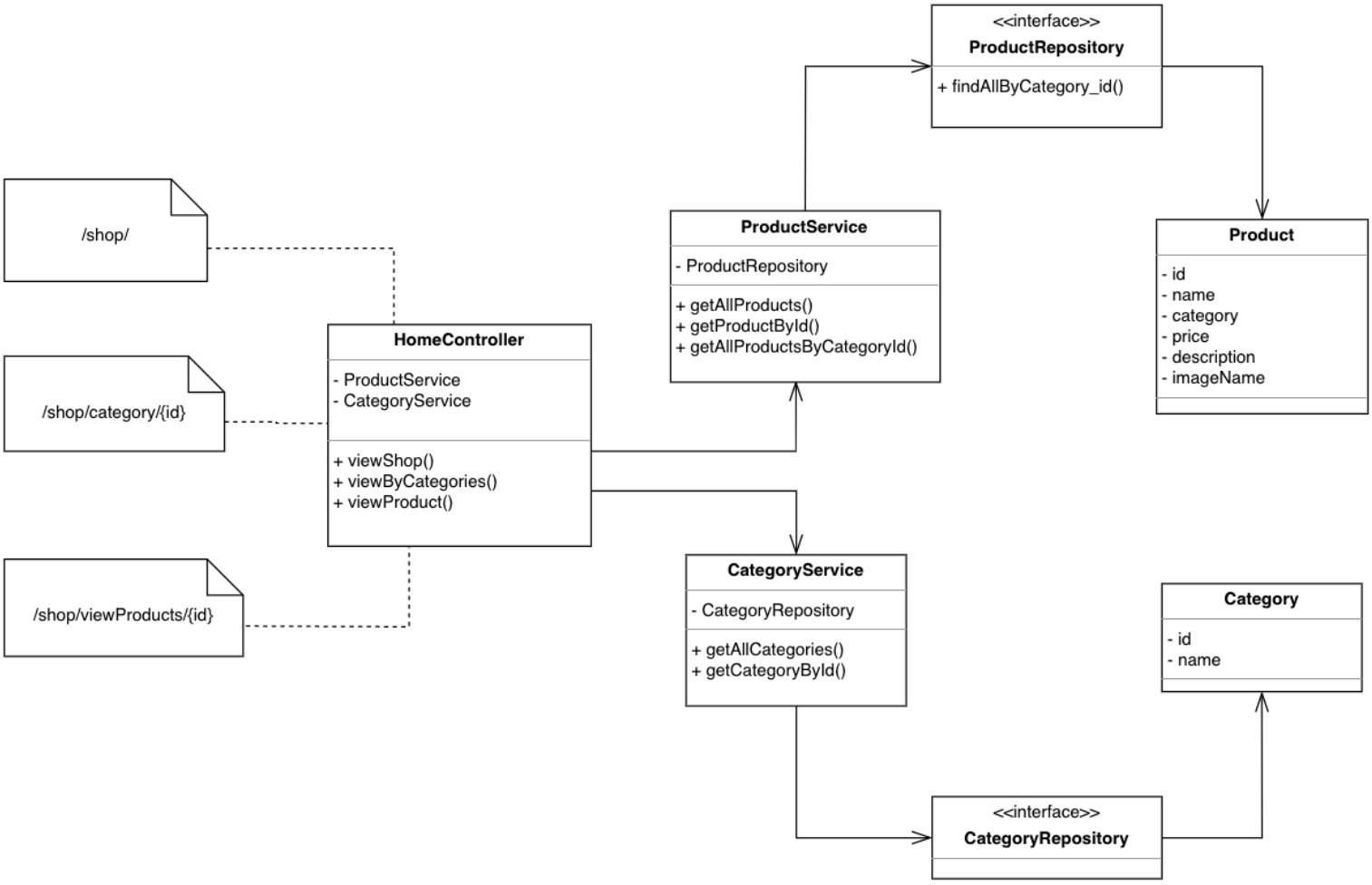

fig2.4.2.1: Class diagram for the view product component

this **View Product** component are "/shop" the view for shop homepage, "/shop/category/{id}" the view for a filter by category id section of the shop and "/shop/viewProduct/{id}" the view for details of a product that we find by id. The ProductService class has defined methods that can retrieve information of Product objects that have been stored in persistent storage. Similarly CategoryService class has defined methods that can retrieve information on Category objects that have been stored in the same database. Both ProductService and CategoryService classes facilitate the logical operations nessecary to serve the Controller class so it can return the requested resources. The image(*fig2.4.2.2*) shows the layered view of the system architecture with our implemented classes.

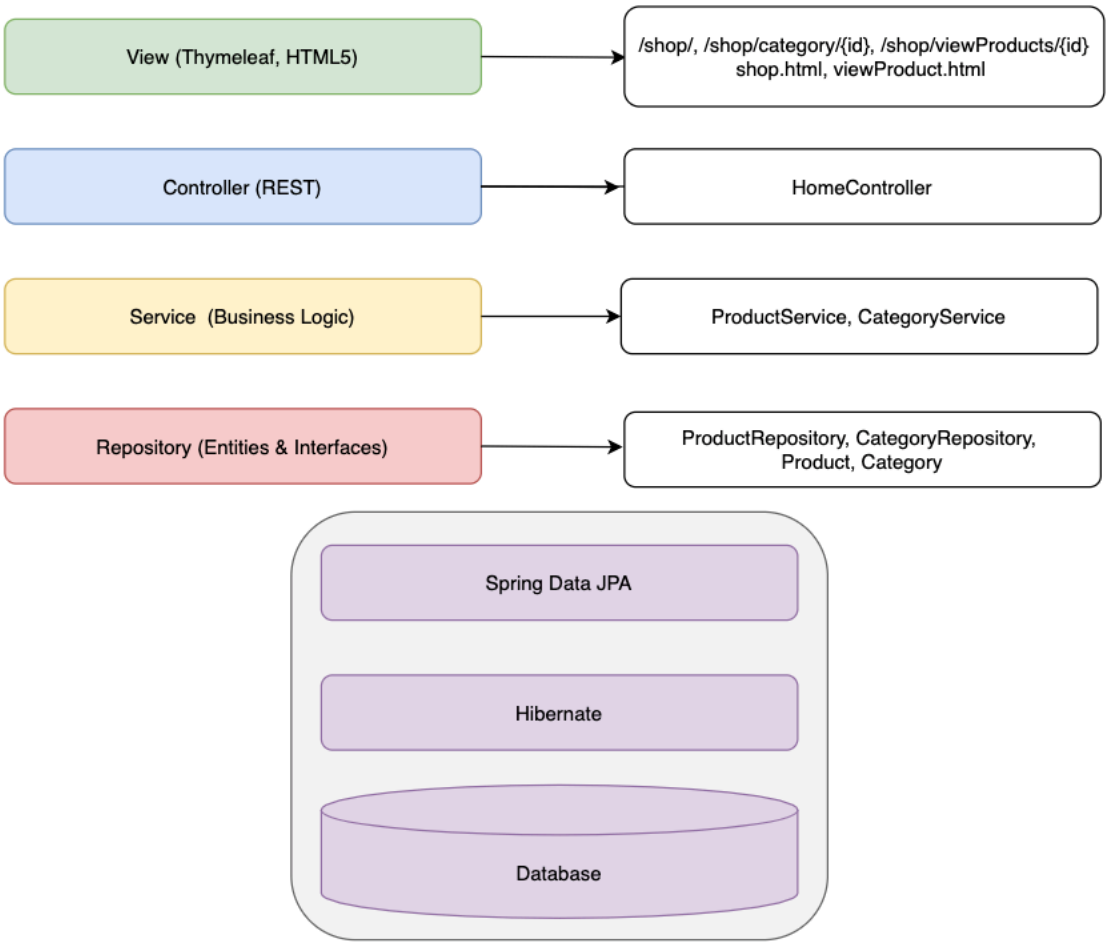

fig2.4.2.2: The view product component in layered system architecture view

2.4.2.1. View Product Component Code

The Product class defines a Product object(*fig2.4.2.1.1*). Using Spring Boot annotations in combination with Spring JPA and Hibernate allows the Product class to be mapped to a product table. The Product class has private member variables of id, name, category, price, description and imageName. Similarly the product table has a primary key of product_id, attributes of name, category, price, description and imageName.

```
@Entity
@Table(name = "product")
public class Product {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    private String name;

    @ManyToOne(fetch = FetchType.LAZY)
    @JoinColumn(name = "category_id", referencedColumnName = "category_id")
    private Category category;

    private float price;
    private String description;
    @Column(name = "image_Name")
    private String imageName;
```

fig2.4.2.1.1: Product.java class

A Category class defines a Category object(*fig2.4.2.1.2*). The Category class has private member variables of id, name that correspond to a category table with primary key of category_id and attribute name. The relationship between the tables “ManyToOne” annotation is also defined in the Product class. During the database design, it was decided that the business rule that many products can belong to one category, here that relationship is implemented. The relevant constructors, getters and setters were each added to the Product and Category classes.

```java
@Entity
@Table(name = "category")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "category_id")
    private int id;

    private String name;
```

fig2.4.2.1.2: Category.java class

A ProductRepository interface(*fig2.4.2.1.3*) and a CategoryRepository interface(*fig2.4.2.1.4*) that both extend JpaRepository are defined. The benefits of extending JpaRepository are that it allows access to CRUD methods that will be used to manipulate/retrieve data in product and category tables. A method findAllByCategory_Id() that finds all products by their id is declared.

```java
import com.github.simplefocals.model.Product;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

import java.util.List;

@Repository
public interface ProductRepository extends JpaRepository<Product, Integer> {
    //method that is made by Jpa create
    List<Product> findAllByCategory_Id(Integer id);
}
```

fig2.4.2.1.3: ProductRepository.java Interface

```java
import com.github.simplefocals.model.Category;
import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.stereotype.Repository;

@Repository
public interface CategoryRepository extends JpaRepository<Category, Integer> {

}
```

fig2.4.2.1.4: CategoryRepository.java Interface

A ProductService class is defined(*fig2.4.2.1.5*). A ProductRepository dependancy is injected in order to facilitate the creation of methods getAllProducts(), getProductById() and getAllProductsByCategoryId() that handle the application logic. Similarly a CategoryService class is defined(*fig2.4.2.1.6*) and in the same way the CategoryRepository aids in the creation of getAllCategories() and getCategoryById() in order to process the business logic.

```java
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    public List<Product> getAllProducts() { return productRepository.findAll(); }

    public void addProduct(Product product){
        productRepository.save(product);
    }

    public void deleteProductById(Integer id) { productRepository.deleteById(id); }

    public Optional<Product> getProductById(Integer id) { return productRepository.findById(id); }

    public List<Product> getAllProductsByCategoryId(Integer id) { return productRepository.findAllByCategory_Id(id); }

}
```

fig2.4.2.1.5: ProductService.java class

```java
@Service
public class CategoryService {

    @Autowired
    private CategoryRepository categoryRepository;

    public void addCategory(Category category) { categoryRepository.save(category); }

    public List<Category> getAllCategories() { return categoryRepository.findAll(); }

    public Optional<Category> getCategoryById(int id) { return categoryRepository.findById(id); }

    public void deleteCategoryById(int id) { categoryRepository.deleteById(id); }
}
```

fig2.4.2.1.6: CategoryService.java class

In the HomeController there are mappings for REST endpoints for this particular service. The image(*fig2.4.2.1.7*) outlines a get request method for the shop which maps to the “/shop” URL and returns the “shop.html”. In the HomeController, the CategoryService and ProductService are used to retrieve data from persistent storage. A model object is also passed in order use the addAttribute() method(*fig2.4.2.1.8*). The addAttribute method in the model class is used to pass a JSON object of a List of categories “categories” and JSON object of a List of products, “products”, to the view.

```
@GetMapping("/shop")
public String viewShop(Model model){
```

fig2.4.2.1.7: HomeController.java The shop mapping

```
    model.addAttribute( s: "categories", categoryService.getAllCategories());
    model.addAttribute( s: "products", productService.getAllProducts());
    return "shop";
}
```

fig2.4.2.1.8: HomeController.java Injecting objects to the model view

The image(*fig2.4.2.1.9*) outlines a get request method for view by category which maps to the “/shop/category/{id}” URL and returns the “shop.html” with filtered categories. In the Home-Controller, the CategoryService and ProductService are used to retrieve data from persistent storage. The {id} is the dynamic element used in deciding which category should be displayed. The “@PathVariable” annotation allows an id element to be taken from the view and facilities the category to be filtered by that specific id. A model object is also passed in order use the addAt-tribute method(*fig2.4.2.1.10*). Similar to the viewShop(), the addAttribute method in the model class is used to pass a JSON object of a List of categories “categories” and JSON object of a List of products, “products”, to the view.

```
@GetMapping("/shop/category/{id}")
public String viewByCategories(@PathVariable int id, Model model){
```

fig2.4.2.1.9: HomeController.java The view by category

```
    model.addAttribute( s: "categories", categoryService.getAllCategories());
    model.addAttribute( s: "products", productService.getAllProductsByCategoryId(id));
    return "shop";
}
```

fig2.4.2.1.10: HomeController.java Injecting objects to the model view

The image(*fig2.4.2.1.11*) outlines a get request method for view product which maps to the "/shop/viewproduct/{id}" URL and returns the "viewproduct.html". In the HomeController, the CategoryService and ProductService are used to retrieve data from persistent storage. The {id} is the dynamic element used in deciding which product should be displayed. The "@PathVariable" annotation allows an id element to be taken from the view and facilities the function of which product should be displayed. A model object is also passed in order use the addAttribute method(*fig2.4.2.1.12*).. The addAttribute method in the model class is used to send a JSON Product object "product" to the view.

```
@GetMapping("/shop/viewproduct/{id}")
public String viewProduct(@PathVariable int id, Model model){
```

fig2.4.2.1.11: HomeController.java The viewproduct mapping

```
    model.addAttribute( s: "product", productService.getProductById(id).get());
    return "viewProduct";
}
```

fig2.4.2.1.12: HomeController.java Injecting Product object to the model view

Using Thymeleaf templating engine we can embed dynamic elements in the HTML5 code for rendering. The image(*fig2.4.2.1.13*) shows a block of code with these dynamic elements insert-ed. A for each loop is used to list each category object from the “categories” previously defined GetMapping viewCategories() method. The category names are then used as links in the shop.html page. The result is this code is in image(*fig2.4.2.1.14*).

```
<div class="col-lg-3">
    <h1 class="h2 pb-4">Categories</h1>
    <ul class="list-group shop-top-menu pb-3 pt-1">
        <a class="h4 text-dark text-decoration-none mr-3" th:href="@{/shop}">
            <li class="list-group-item ">All Products</li>
        </a>
        <a class="h4 text-dark text-decoration-none mr-3" href="" th:each="category : ${categories}"
           th:href="@{/shop/category/{id}(id=${category.id})}">
            <li class="list-group-item" th:text="${category.name}"></li>
        </a>
    </ul>
</div>
```

fig2.4.2.1.13: Shop.html Thymeleaf embedded code

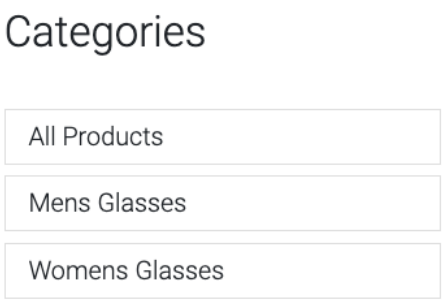

fig2.4.2.1.14: Shop.html listed categories

The image(*fig2.4.2.1.15*) shows code for the listing of products in shop.html page. Similarly as before a for each loop is used to list the products on the page. The “products” JSON object defined in the GetMapping viewShop() method is accessed here. The name and price of each object is displayed on cards in the rendered front-end pages(*fig2.4.2.1.16*).

```
<!-- Start of One Card -->
<div class="col-md-4" th:each="product : ${products}">
    <div class="card mb-4 product-wap rounded-0" >
        <div class="card rounded-0">
            <img class="card-img" height="300px" th:src="@{${'/productImages/' + product.imageName}}"
                th:alt="${product.name}">
            <a href="#" th:href="@{/shop/viewproduct/{id}(id=${product.id})}">
                <div class="card-img-overlay rounded-0 product-overlay d-flex align-items-center
                justify-content-center"></div>
            </a>
        </div>
        <div class="card-body">
            <p class="h1 py-2" th:text="${product.name}"></p>
            <p class="h3 py-2" >
                € <span th:text="${#numbers.formatDecimal(product.price,  1, 'COMMA', 2, 'POINT')}"></span>
            </p>
        </div>
    </div>
</div>
<!-- End of One Card -->
```

fig2.4.2.1.15: Shop.html card snippet for the rendering of products

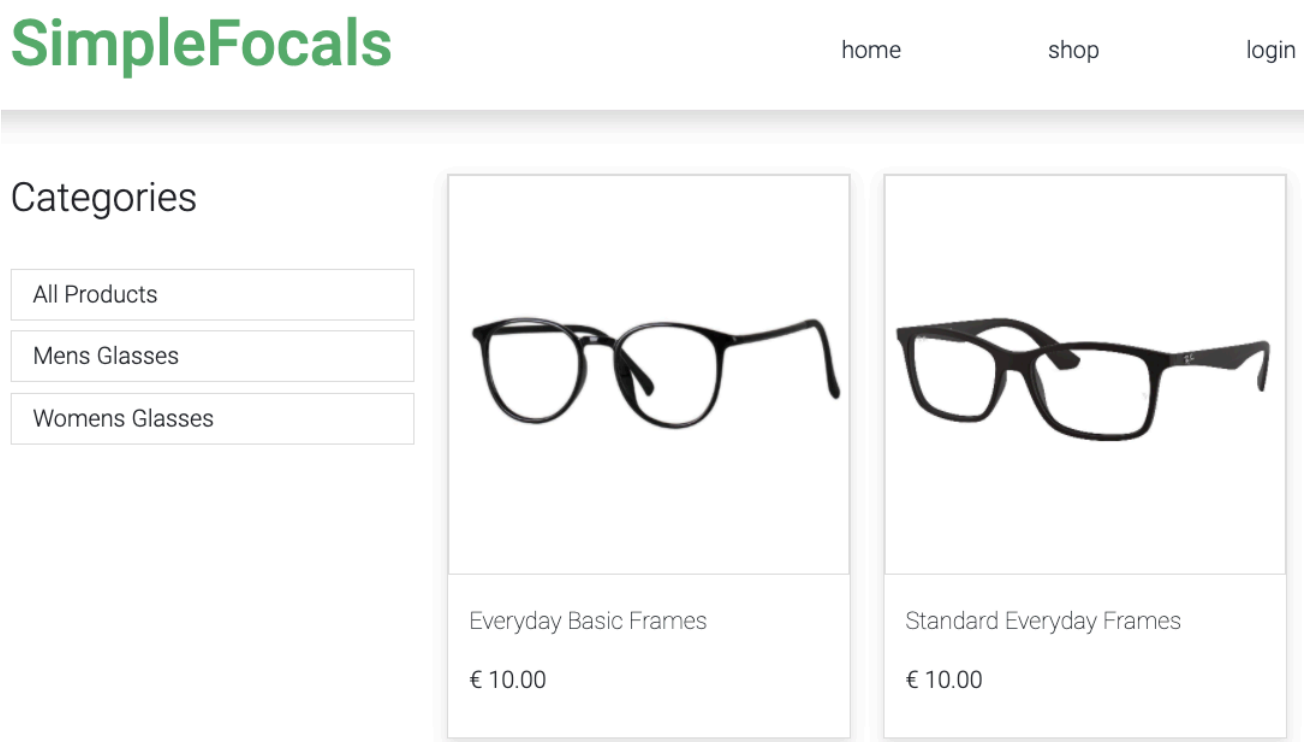

fig2.4.2.1.16: Shop.html products being displayed

The image(*fig2.4.2.1.17*) shows a code snippet from the individual product page in viewproduct.html page. The “products” JSON object defined in the GetMapping viewProduct() method, that product object with a specific id is accessed here. The name, the category, the description and price of that product object with specific id is displayed on card in the rendered front-end page(*fig2.4.2.1.18*).

```
<div class="card-body">

    <h1 class="h2 text-center" th:text="${product.name}"></h1>
    <br>
    <p class="h3 py-2"><strong>Category:</strong></p>
    <p class="h4 py-2" th:text="${product.category.name}"></p>

    <p class="h3 py-2"><strong>Price:</strong></p>
    <p class="h4 py-2">€
        <span th:text="${#numbers.formatDecimal(product.price,  1, 'COMMA', 2, 'POINT')}">
    </span></p>
    <p class="h3 py-2"><strong>Description:</strong></p>
    <p class="h4 py-2" th:text="${product.description}"></p>
```

fig2.4.2.1.17: viewProduct.html product attributes being accessed

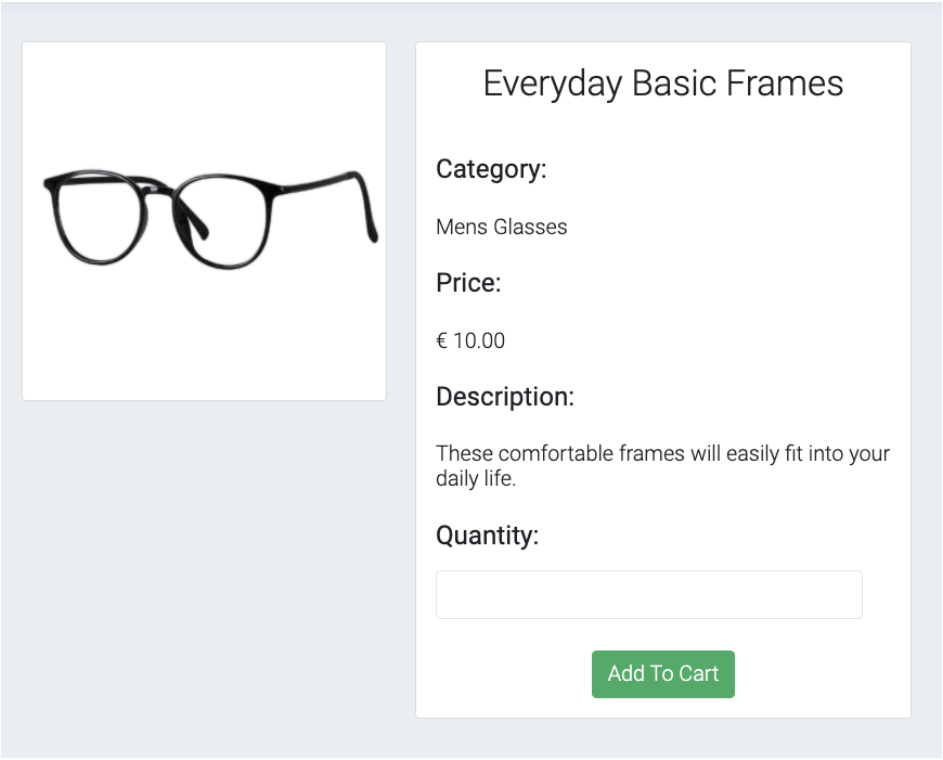

fig2.4.2.1.18: viewProduct.html full product view

2.4.3. Register Component.

This section describes the register component of the application. A non-registered customer enters a first name, a last name, a unique email and secure password to a form and then pushes the submit button to register with the system. The system saves the customers details and completes the registration by redirecting to the login page. The image(*fig2.4.3.1*) shows the class diagram of the register function of the system. The image(*fig2.4.2.2*) shows how the classes fit into the application layered architecture.

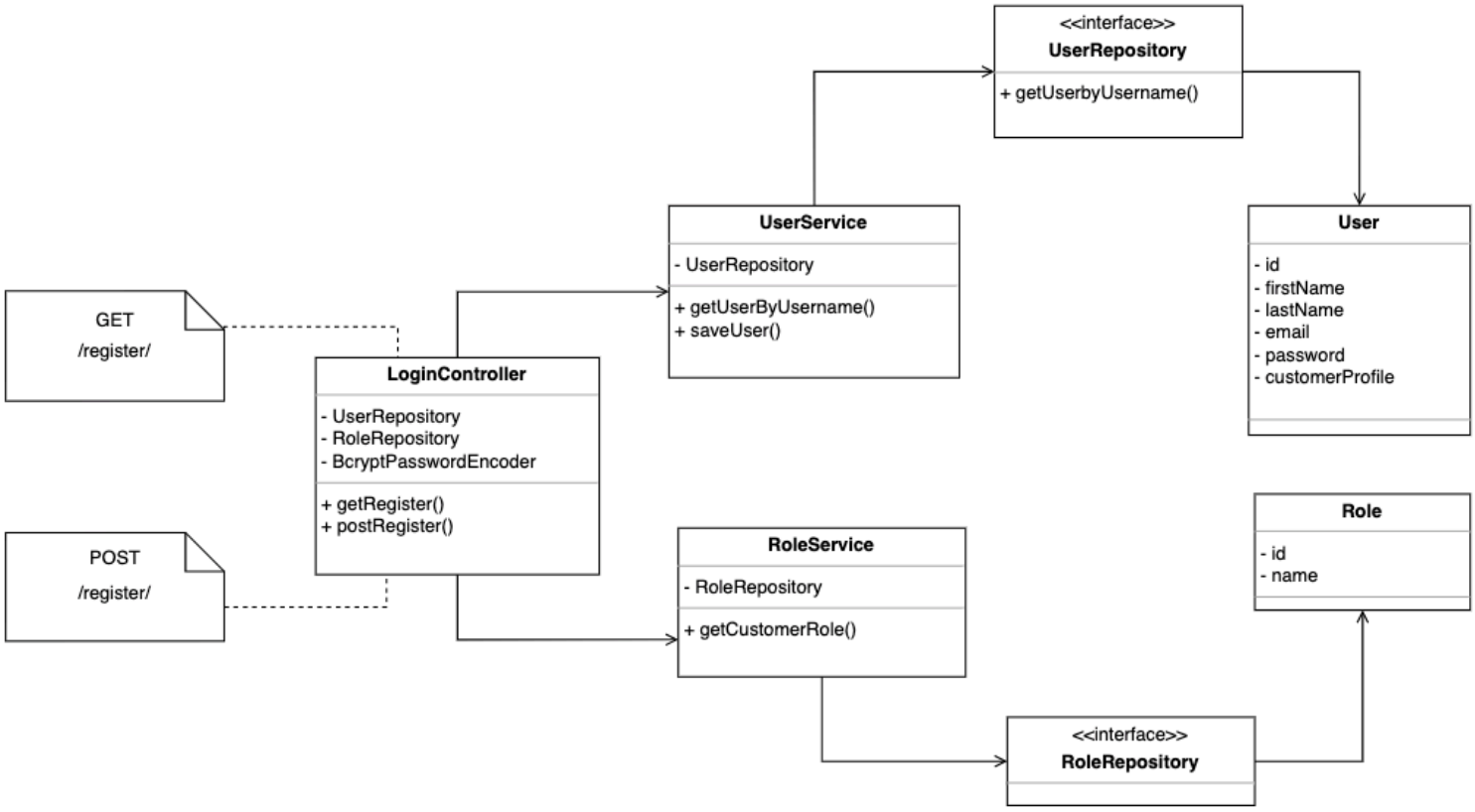

fig2.4.3.1: Class diagram for the view product component

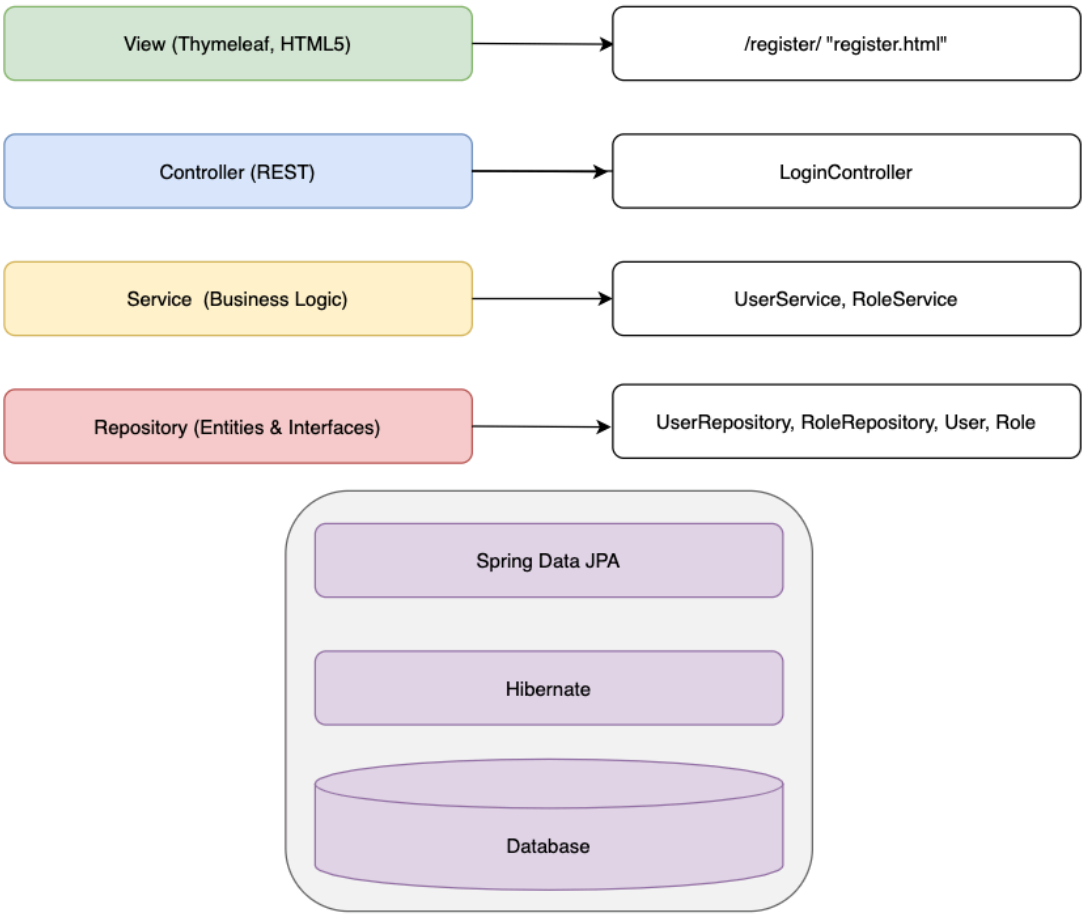

fig2.4.3.2: The register component in layered system architecture view

2.4.3.1. Register Component Code.

A User class(*fig2.4.3.1.1*) and Roles class(*fig2.4.3.1.2*) are defined. The User class describes a User, that has attributes of id, firstName, lastName, email, password and customerProfile. The customerProfile is default set to null when a user is created(*only when a customer registers and logs into the system a customerProfile can be created, this is outside the scope of this component*). The The Role class defines what type of role a user has, a role has attributes of id and name. The relationship between the two classes is a “ManyToMany” relationship. This means that a user can have many roles and a role can be fulfilled by many users. Similar to previous, in the ***2.4.2. View Product Component*** implementation, these classes will be mapped to tables in the database and their relationship with each other defined. For each class the relevant constructors, getters and setters were implemented.

```java
@Entity
@Table(name = "users")
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "user_id")
    private Integer id;

    @ManyToMany(cascade = CascadeType.MERGE, fetch = FetchType.EAGER)
    @JoinTable(
            name = "users_roles",
            joinColumns = @JoinColumn(name = "user_id"),
            inverseJoinColumns = @JoinColumn(name = "role_id")
    )
    private List<Role> roles;

    @Column(name = "email", nullable = false, unique = true, length = 255
    @NotEmpty(message = "email may not be empty")
    @Email(message = "{errors.invalid_email}")
    private String email;

    @NotEmpty(message = "name may not be empty")
    private String firstName;
    private String lastName;


    @NotEmpty(message = "password may not be empty")
    private String password;

    @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL)
    private CustomerProfile customerProfile;
```

fig2.4.3.1.1: User.java class

```java
@Entity
@Table(name = "roles")
public class Role {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "role_id")
    private Integer id;

    @Column(nullable = false, unique = true)
    @NotEmpty
    private String name;

    @ManyToMany(mappedBy = "roles")
    private List<User> users;
```

fig2.4.3.1.2: Role.java class

Two interfaces classes that extend JpaRepository, UserRepository(*fig2.4.3.1.3*) and RoleRepository(*fig2.4.3.1.4*) are defined so that CRUD operations can be accessed in the service layer when defining the business logic. In UserRepository a query to the database was written specifically to retrieve a user by email.

```
public interface UserRepository extends JpaRepository<User, Integer> {

    @Query("SELECT u FROM User u WHERE u.email = :email")
    User getUserByUsername(@Param("email") String email);

}
```

fig2.4.3.1.3: UserRepository.java interface

```
public interface RoleRepository extends JpaRepository<Role, Integer> {
}
```

fig2.4.3.1.4: RoleRepository.java interface

Two Service classes, UserService(*fig2.4.3.1.5*) and RoleService(*fig2.4.3.1.6*) are defined. UserService has a getUserByUsername() that was previously defined in the UserRepository class and a saveUser() method, that is used to write a customer to the database. The RoleService class has one method, getCustomerRole() this method is used to return the role at role_id = 2 in the roles table. The roles table in the database has two entries, one that defines an ADMIN and another that defines a CUSTOMER(*fig2.4.3.1.7*).

	role_id	name
1	1	ADMIN
2	2	CUSTOMER

fig2.4.3.1.7: Roles table

```java
@Service
public class UserService {
Type to enter text

    @Autowired
    private UserRepository userRepository;

    //the username is email
    public User getUserByUsername(String email) { return userRepository.getUserByUsername(email); }

    public User saveUser(User user) { return userRepository.save(user); }
}
```

fig2.4.3.1.5: UserService.java class

```java
@Service
public class RoleService {

    @Autowired
    private RoleRepository roleRepository;

    public Role getCustomerRole() { return roleRepository.findById(2).get(); }
}
```

fig2.4.3.1.6: RoleService.java class

In the LoginController there are mappings for REST API endpoints for the register service. The image(*fig2.4.3.1.7*) outlines a get request method for register which maps to the "/register" URL and returns the "register.html". In the LoginController, the UserService, RoleService and a BcryptPasswordEncoder private variables are autowired in the class(*fig2.4.3.1.8*).

```java
@GetMapping("/register")
public String getRegister() { return "register"; }
```

fig2.4.3.1.7: LoginController.java register mapping

```
@Autowired
private UserService userService;

@Autowired
private RoleService roleService;

@Autowired
private BCryptPasswordEncoder bCryptPasswordEncoder;
```

fig2.4.3.1.8: LoginController.java private member variables

The BcryptPasswordEncoder object is used to encode the password that is inputted into the register.html form by the non-registered customer. The password is then encoded/hashed and that value is then saved to the database. This is done as a security precaution because saving unencoded password to the database is deemed unsafe practice. Since the password is encoded, if there is a situation where user information is leaked it does not compromise a customers login to the system.

The image(*fig2.4.3.1.9*) shows the post method when the form on “register.html” is submitted. The “@ModelAttribute(“user”) User user” passed in the method parameters allows the front-end to pass a “user” JSON Object to User object in the back-end. In this method, a user with specific unique email is checked. If the user exists we throw and exception. If such a user doesn’t exist the user password is encoded using BcyptPasswordEncoder. The method retrieves the role of customer and assigns it to the user. Finally the user is saved to the system and the user is redirected to the login page. The image(*fig2.4.3.1.10*) shows a non-registered customer fill out the fields of the form. Once they enter details with unique email address and press the submit button, if there are no errors the system redirects them to the login page. With successful registration the system adds a new entry to the User table.

```java
@PostMapping("/register")
public String postRegister(@ModelAttribute("user") User user) throws UserExistsException {
    //if the User is not registered on the system
    if(userService.getUserByUsername(user.getEmail()) == null){
        // assign the user's password to String
        String rawData = user.getPassword();
        //set the users password to the encoded value
        user.setPassword(bCryptPasswordEncoder.encode(rawData));

        //make a Roles List object
        List<Role> roles = new ArrayList<>();
        //add this customer to the roles
        roles.add(roleService.getCustomerRole());
        //set the users role to customer
        user.setRoles(roles);
        //write the user to persistent storage
        userService.saveUser(user);
        //redirect to login
        return "redirect:/login";
```

fig2.4.3.1.9: LoginController.java Post method when the form is filled in and sent.

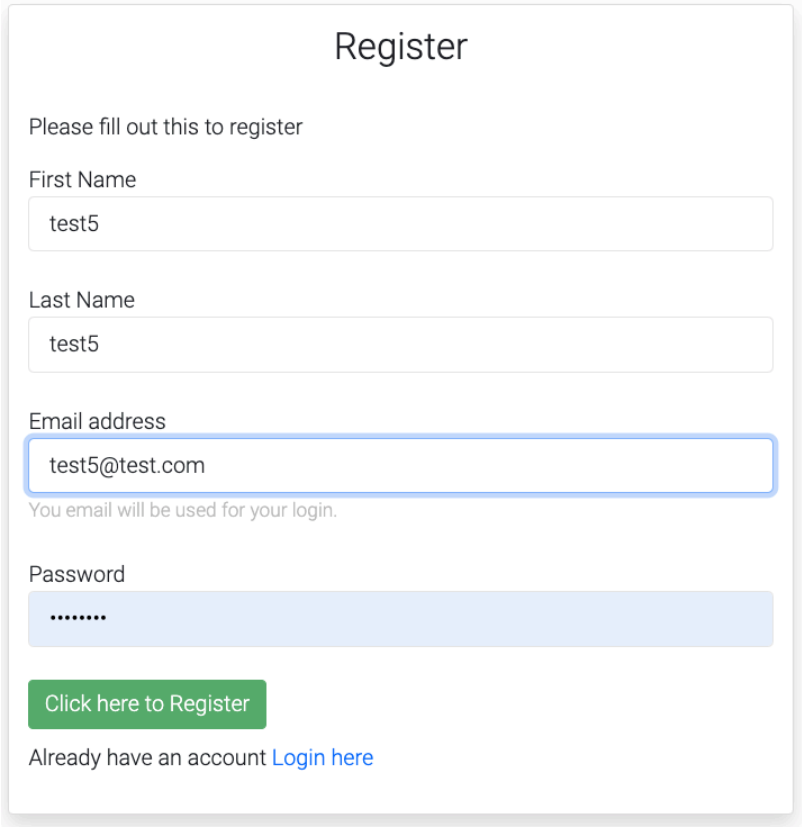

fig2.4.3.1.10: register.html register form

The image(*fig2.4.3.1.11*) shows that the Customer has registered as a User in the database.

	user_id	email	first_name	last_name	pass...	customer_profile_id
1	1	christobelchan@gmail.com	chris	chan	$2a$10$AVWm…	<null>
2	2	test@test.com	test	test	$2a$10$CpO5…	1
3	17	test1@test.com	test1	test1	$2a$10$2qUP…	37
4	18	test2@test.com	test2	test2	$2a$10$7OTi…	38
5	39	test3@test.com	test3	test3	$2a$10$RTjl…	40
6	41	test5@test.com	test5	test5	$2a$10$3PJq…	<null>

fig2.4.3.1.11: The user table

2.4.4. Login Component.

This section describes the login component of the system. When a user navigates to the login page and enters their email and password combination, the system logs them into the application. When a customer that has previously registered with the system logs in they will be directed to the homepage of the web application. When an administrator logs in they will be directed to the CMS homepage. The image shows the class diagram(*fig2.4.4.1*) for the login component.

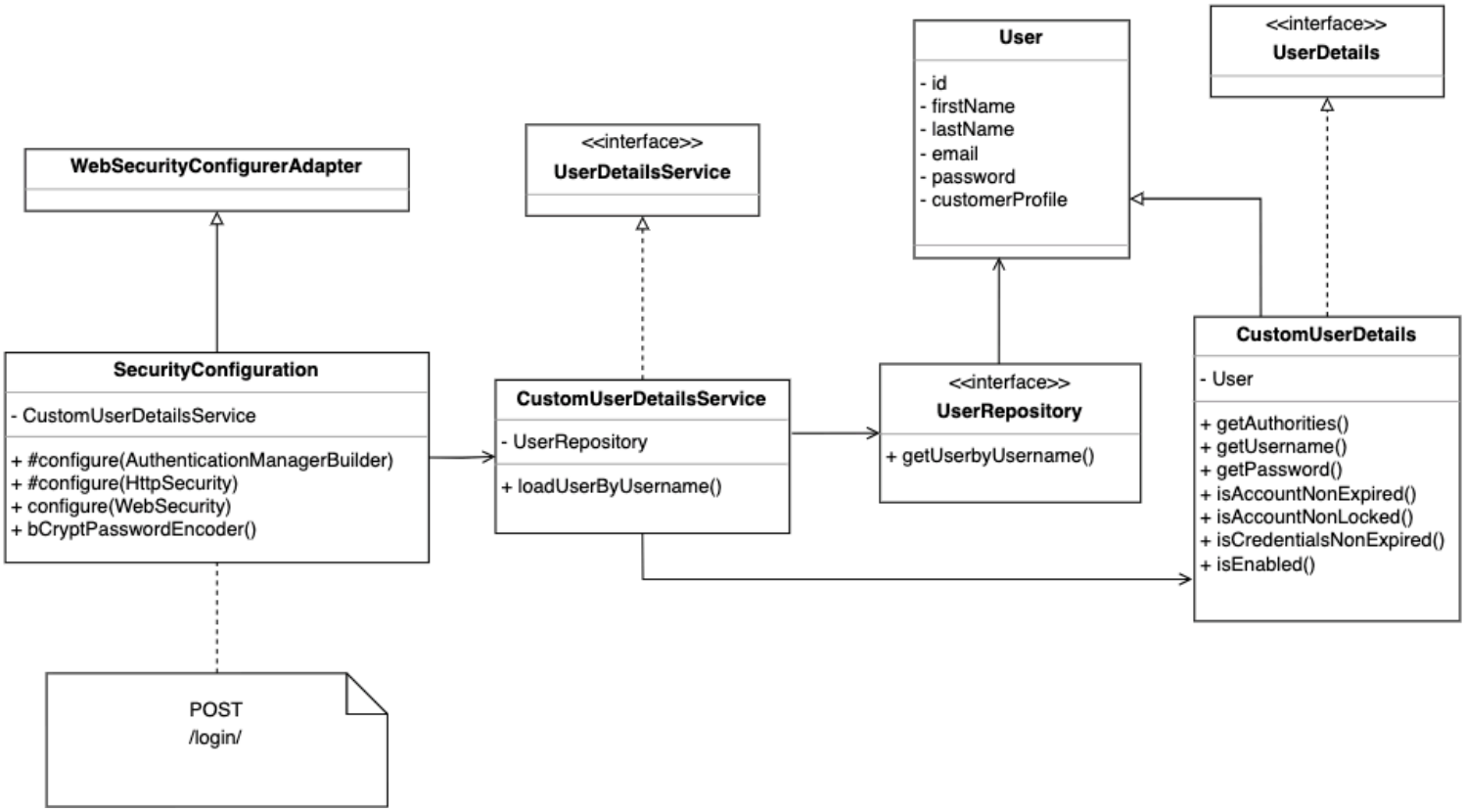

fig2.4.4.1: Class diagram for the login component

2.4.4.1. Login Component code

SecurityConfiguration.java file handles the authorisation and authentication throughout the application. In Spring security an AuthenticationManger is the mechanism that manages authentication in Spring applications. Implementation of authentication is conducted by configuring a AuthenticationManagerBuilder and passing it properties. Once this is done a AuthenticationManager is created and authenticates based on the given configuration. The image(*fig2.4.4.1.1*) shows the class SecurityConfiguration. By extending WebSecurityConfigurerAdapter, configure methods are able to be overridden that allow authentication and authorisation configurations.

```
@Configuration
@EnableWebSecurity
public class SecurityConfiguration extends WebSecurityConfigurerAdapter {
```

fig2.4.4.1.1: SecurityConfiguration.java class definition

The image(*fig2.4.4.1.2*) shows a configure method for authentication. The method takes an AuthenticationManagerBuilder object in the parameters and allows for us to specify on what properties to authenticate with. The AuthenticationManagerBuilder object has a method userDetailsService that allows us to authenticate based on an instance of userDetailsService. In the application we created a custom implementation of this called CustomUserDetailService, that was made into a Spring service. Now with the configuration for AuthenticationManager, anytime there is an authentication attempt Spring security is going to call the method on CustomUserDetailService to get user information in order to authenticate.

```
@Override
protected void configure(AuthenticationManagerBuilder auth) throws Exception {
    auth.userDetailsService(customUserDetailsService);
}
```

fig2.4.4.1.2: SecurityConfiguration.java configuring the AuthenticationManagerBuilder

The image(*fig2.4.4.1.3*) shows bCryptPasswordEncoder method. This will be used in order to validate the password corresponding to the user details that are retrieved when authentication is attempted.

```
@Bean
public BCryptPasswordEncoder bCryptPasswordEncoder(){
    return new BCryptPasswordEncoder();
}
```

fig2.4.4.1.3: SecurityConfiguration.java
Security Configuration method bCryptPasswordEncoder

The image(*fig2.4.4.1.4*) shows configuration for the specific static resources to be accessible throughout the application.

```
//allows access to resources directories that contain static content
@Override
public void configure(WebSecurity web) throws Exception {
    web.ignoring().antMatchers( ...antPatterns: "/resources/**"
                                              , "static/**"
                                              , "/images/**"
                                              , "/productImages/**"
                                              , "/prescription/**"
                                              , "/css/**"
                                              , "/js/**");
}
```

fig2.4.4.1.4: SecurityConfiguration.java
Configuration for websecurity

The image(*fig2.4.4.1.5*) shows authorisation for specific resources and what level of user can access them. From the image we can see that unauthenticated users of the system can access the homepage, the shop and its subdirectories, the register and the login pages. A registered

customer can access the profile page and subdirectories, the create payment intent page and the payment page. The administrator has access to admin page with all the subdirectories.

```
//authorisation of web end-points
@Override
protected void configure(HttpSecurity http) throws Exception {

    http.authorizeRequests() ExpressionUrlAuthorizationConfigurer<...>.ExpressionInterceptUrlRegistry
            //The following endpoints are accessible to unauthenticated users
        .antMatchers( ...antPatterns: "/").permitAll()
        .antMatchers( ...antPatterns: "/shop/**").permitAll()
        .antMatchers( ...antPatterns: "/register").permitAll()
            //The following endpoints are accessible to authenticated Customer
            .antMatchers( ...antPatterns: "/profile/**").hasAuthority("CUSTOMER")
            .antMatchers( ...antPatterns: "/create-payment-intent").hasAuthority("CUSTOMER")
            .antMatchers( ...antPatterns: "/payment").hasAuthority("CUSTOMER")
            .antMatchers( ...antPatterns: "/profile/**").hasAuthority("CUSTOMER")
            //The following endpoints are accessible to authenticated Admin
        .antMatchers( ...antPatterns: "/admin/**").hasAuthority("ADMIN")
        .anyRequest().authenticated()
```

fig2.4.4.1.5: SecurityConfiguration.java Http configurations

The image(*fig2.4.4.1.6*) shows http security authorisation specification for the login. The application shall use its own login page. It will take the fields of email as the username and the password.

```
.formLogin() FormLoginConfigurer<HttpSecurity>
.loginPage("/login").permitAll()
.defaultSuccessUrl("/")
.usernameParameter("email")
.passwordParameter("password")
```

fig2.4.4.1.6: SecurityConfiguration.java Http configurations

The image(*fig2.4.4.1.7*) shows the CustomUserDetailsService.java file. This class implements UserDetailsService. This custom class was created to facilitate retrieving the user from the database so it can be added for authorisation configuration in the SecurityConfiguration class. If the customer exists we return a new CustomUserDetails object of that user. If not an exception is thrown.

```
@Service
public class CustomUserDetailsService implements UserDetailsService {

    //so we can retrieve the user from the db
    @Autowired
    private UserRepository userRepository;

    //method to get the user by email
    @Override
    public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {

        User user = userRepository.getUserByUsername(email);
        if(user == null){
            throw new UsernameNotFoundException("The email does not exist");
        }
        return new CustomUserDetails(user);
    }
```

fig2.4.4.1.7: CustomUserDetailsService.java

The image(*fig2.4.4.1.8*) shows the CustomUserDetails.java file. This class is needed because the CustomUserDetailsService class requires an instance of CustomUserDetails to pass to the SecurityConfiguration for authorisation. This class implements UserDetails and has abstract methods that need to be overridden. There is one Constructor that takes an instance of User in the parameters. The first method overridden is getAuthorities(). In the method implementation we add the roles that are saved in the database to the List of grantedAuthorities and then return that List.

```java
public class CustomUserDetails extends User implements UserDetails {

    private User user;

    public CustomUserDetails(User user) { this.user = user; }

    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        //make a new grantedAuthorityList
        List<GrantedAuthority> grantedAuthorityList = new ArrayList<>();
        //make a List of the roles
        List<Role> userRoles = user.getRoles();
        //add each role to the grantedAuthorityList
        for (Role role: userRoles) {
            grantedAuthorityList.add(new SimpleGrantedAuthority(role.getName()));
        }
        // then return grantedAuthority instances
        return grantedAuthorityList;
    }
```

fig2.4.4.1.8: CustomUserDetails.java getAuthorities()

The image(*fig2.4.4.1.9*) shows the methods implementations of getUsername(), that retrieves the user's username and getPassword that returns the user's password.

```java
@Override
public String getUsername() {
    return user.getEmail();
}

@Override
public String getPassword() { return user.getPassword(); }
```

fig2.4.4.1.9: CustomUserDetails.java getUsername(), getPassword()

Thymeleaf parses the fields for email and password on the login.html(*fig2.4.4.1.10*) form and once a post method is submitted SecurityConfiguration deals with the authentication.

```
<form th:action="@{/login}" method="post">
    <div class="form-group">
        <label for="email">Email</label>
        <input type="email" name="email" id="email" placeholder="Email"
               class="form-control form-control-lg">
    </div>
    <br>
    <div class="form-group">
        <label for="password">Password</label>
        <input type="password" class="form-control form-control-lg" placeholder="Password"
               name="password" id="password">
    </div>
```

fig2.4.4.1.10: CustomUserDetails.java getUsername(), getPassword()

A CustomUserDetailsService object passed to the to the AuthenticationManagerBuilder and the user configuration is set. Spring security will verify the credentials for the CustomUserDetails of that user and will allow or deny access based on the entered fields. The image(*fig2.4.4.1.11*) shows the login.html where a customer has entered their login details.

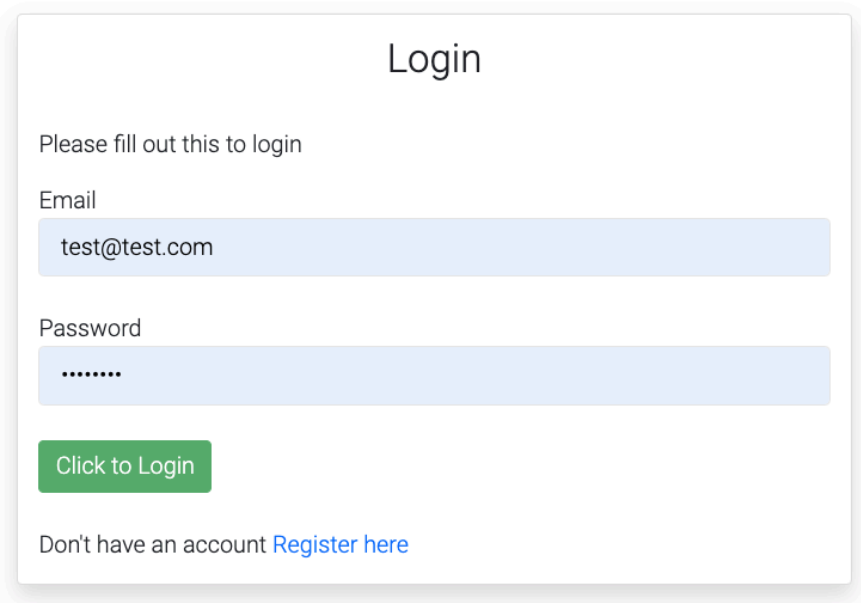

fig2.4.4.1.11: login.html customer login

The image(*fig2.4.4.1.12*) shows that the login was successful and the customer was redirected to a homepage where they have access to cart and profile functionality.

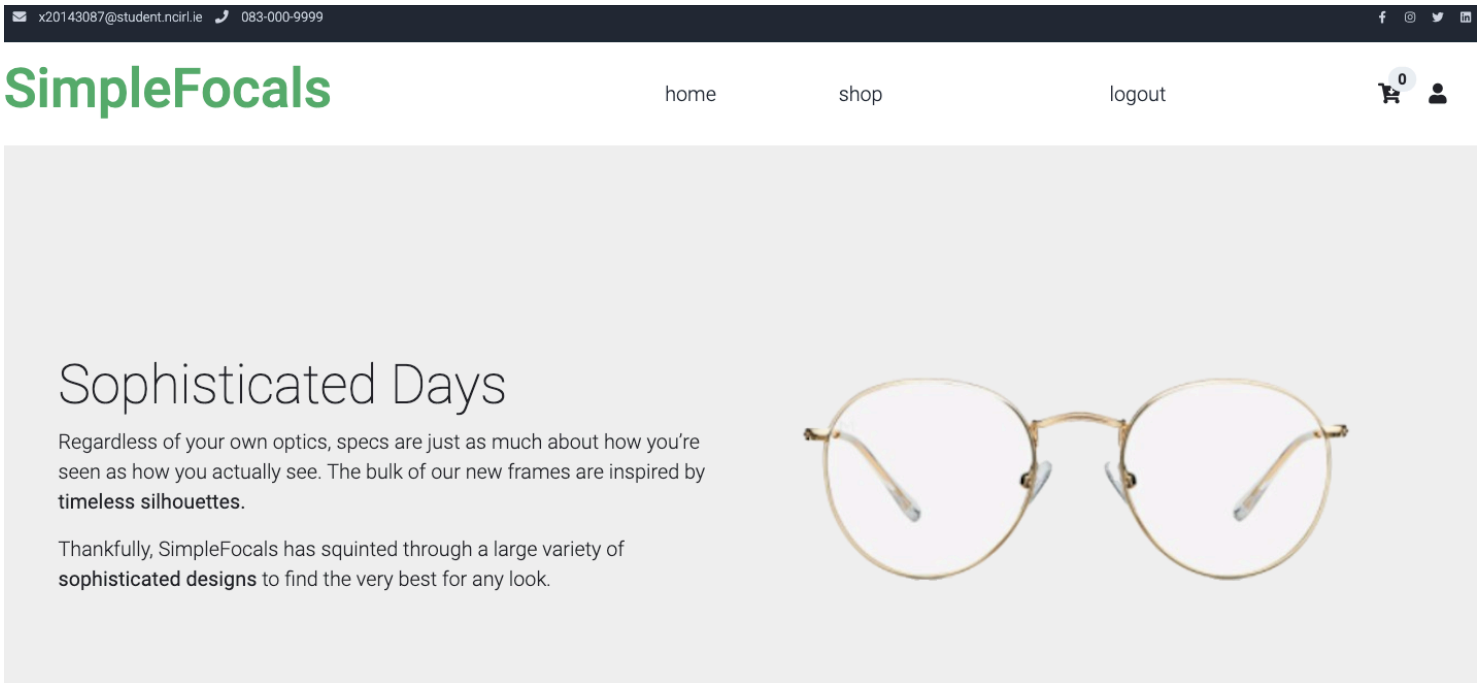

fig2.4.4.1.12: login.html customer successful login

The image shows an administrator entering their details to the login form(*fig2.4.4.1.13*). The administrator is directed to the CMS page when login is successful(*fig2.4.4.1.14*).

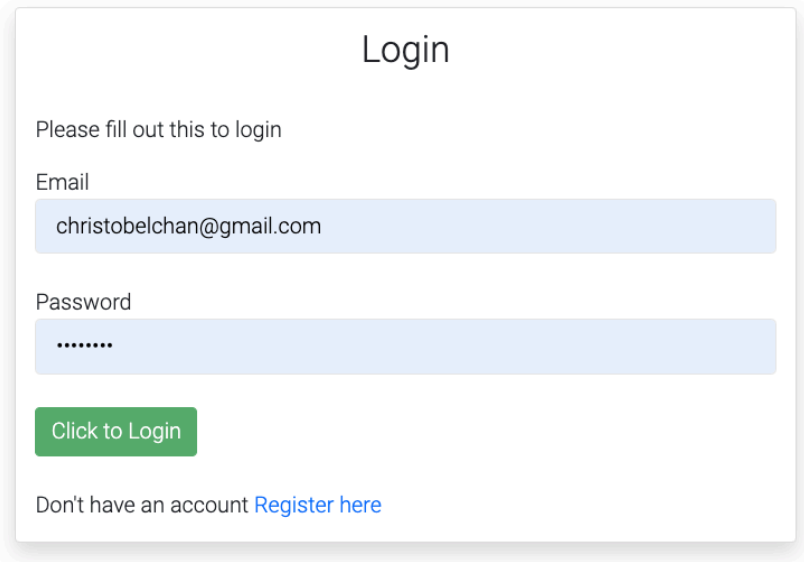

fig2.4.4.1.13: login.html administrator login

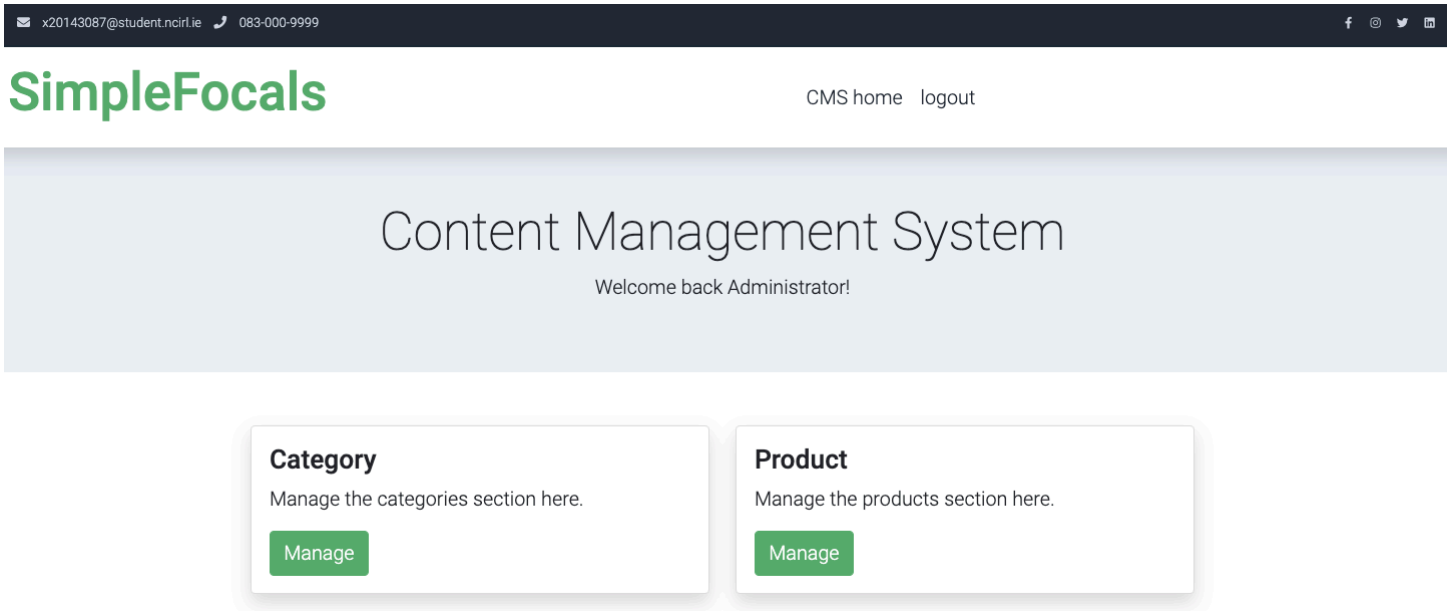

fig2.4.4.1.14: login.html administrator successful login

2.4.5. Add to Cart Component.

This section describes the add to cart component of the application. A registered customer that has a customer profile and has logged into the system has access to a cart. A customer can add to cart on the detailed view of the product. A customer has to enter the quantity of item they want and when they push add to cart button, the items will be added to their cart. The system will redirect them to the shop page and their cart items number will be updated. The image(*fig2.4.5.1*) shows the class diagram of this component. The UserService and the CustomUserDetailsService classes are the same service classes that have been described in ***2.4.3. Register Component*** and ***2.4.4. Login Component*** previously. They are included here in order to facilitate retrieving details of a logged in customer that has made a customer profile. The image(*fig2.4.5.2*) show the layered architectural view of the component.

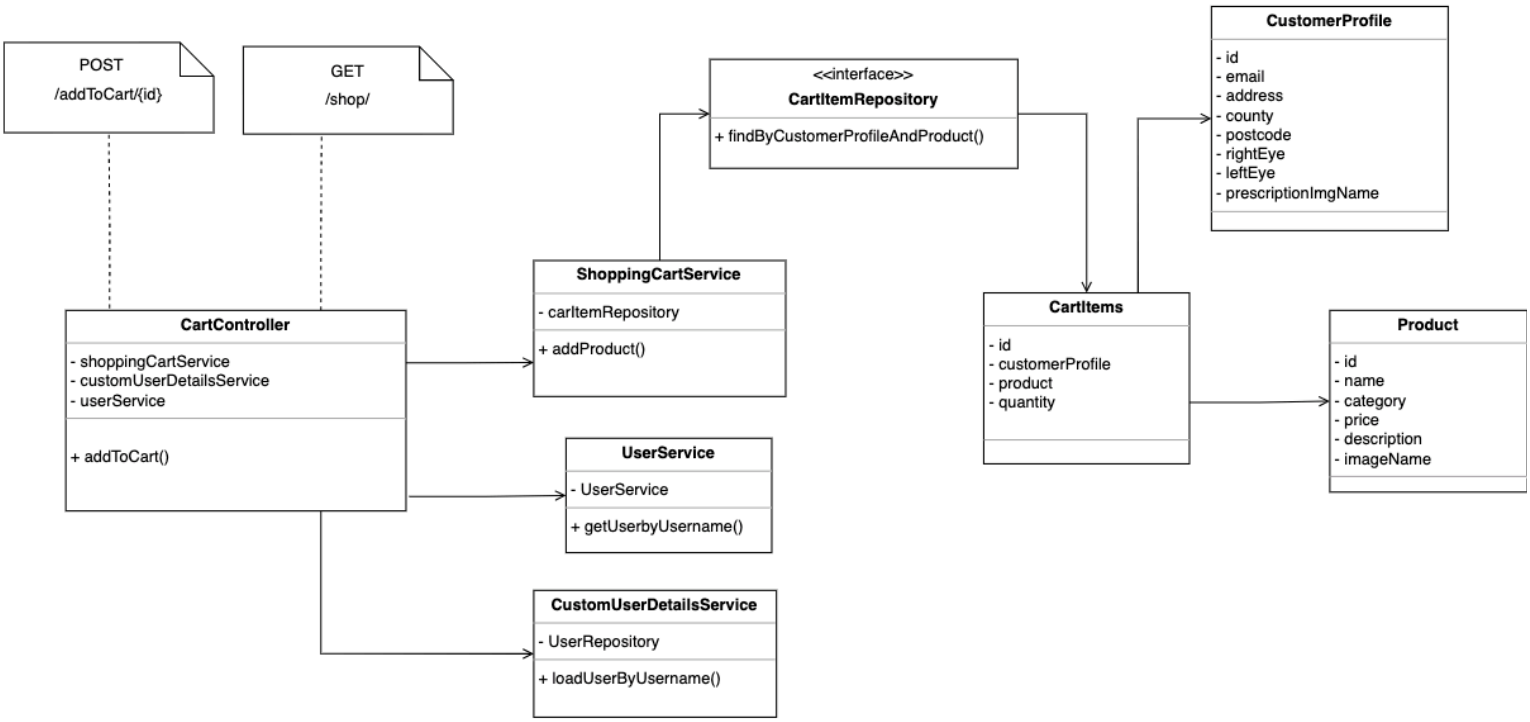

fig2.4.5.1: class diagram for the add to cart component.

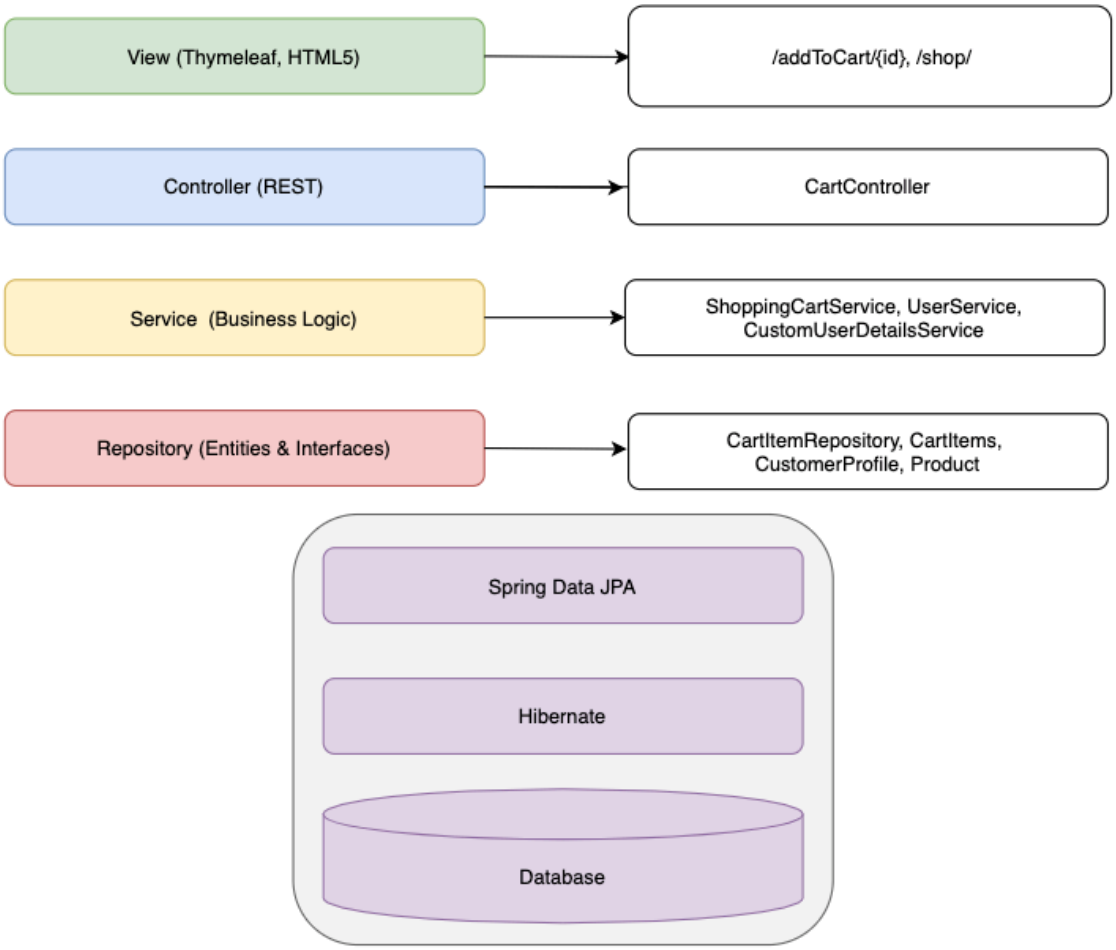

fig2.4.5.2: layered architecture view of the add to cart component.

2.4.5.1. Add to Cart Component code.

The CartItems class was implemented(*fig2.4.5.1.1*). This class has relationships with the CustomerProfile and Product entity/model classes. This is modelled in the class implementation of CartItems, where a CustomerProfile can have many CartItems and CartItems can consist of many Products. The member variables of CartItems are a unique identifier id, a customerProfile, a product and quantity of that product.

```
@Entity
@Table(name = "cart_items")
public class CartItem {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Integer id;

    @ManyToOne
    @JoinColumn(name = "customer_profile_id")
    private CustomerProfile customerProfile;

    @ManyToOne
    @JoinColumn(name = "product_id")
    private Product product;
    private int quantity;
```

fig2.4.5.1.1: CartItems.java

The image(*fig2.4.5.1.2*) shows the CartItemsRepository interface that extends JpaRepository. JpaRepository has implementations of CRUD methods that just require specification. The method we specify is findByCustomerProfileAndProduct(). This method is needed to create a addProduct() method in the service layer.

```
public interface CartItemRepository extends JpaRepository<CartItem, Integer> {

    public CartItem findByCustomerProfileAndProduct(CustomerProfile customerProfile, Product product);
```

fig2.4.5.1.2: CartItemsRepository.java

The image(*fig2.4.5.1.3*) shows the ShoppingCartService class with the implementation of addProduct(). The method checks the database if a cartItems exist of a customerProfile for a specific product using the findByCustomerProfileAndProduct() from CartItemRepository. If a cartItems instance does exist then the value for quantity is just updated for that entry. If it does not exist then a new cartItems object is created with the customerProfile, Product and the quantity of that item and written to the database.

```
@Service
public class ShoppingCartService {

    @Autowired
    private CartItemRepository cartItemRepository;

    public Integer addProduct(Integer quantity, Integer productId, CustomerProfile customerProfile

        //var declarations
        Integer updatedQuantity = quantity;
        Product product = new Product(productId);

        //get the cartitems of a customer for a specific product
        CartItem cartItem = cartItemRepository.findByCustomerProfileAndProduct(customerProfile,
                                                                                    product);
        // if there is no cartitems of a customer for a specific product
        if(cartItem != null){
            //just add the quantity
            updatedQuantity = cartItem.getQuantity() + quantity;
        }else {
            //make a new cartitems object and set the customer profile and product to attributes
            cartItem = new CartItem();
            cartItem.setCustomerProfile(customerProfile);
            cartItem.setProduct(product);
        }

        //set the quantity
        cartItem.setQuantity(updatedQuantity);
        //write to presistent storage
        cartItemRepository.save(cartItem);

        return updatedQuantity;
    }
```

fig2.4.5.1.3: ShoppingCartService.java

The CartController class contains the post mapping for the addToCart() function(*fig2.4.5.1.4*). The @PathVaribale id, indicates which product the customer is going to add to cart. The model also has a Quantity object “quantity” that has a value “value” associated with it. In the function we retrieve the customerProfile of the logged in customer with the help from UserDetailsSerivce and UserService class methods. If the customer hasn’t made a customerProfile yet, the method redirects to “profile.html”, if not details of that customerProfile are retrieved. The function then takes the value inputted from the “value” variable that is quantity the customer has selected from he front-end. The shoppingCartService then adds the product with the fields for value and, the product id and the customerProfile and writes it to the database. The method resolves redirecting to the shop.

```
@PostMapping("/addToCart/{id}")
public String addToCart(@PathVariable int id,
                        @ModelAttribute("quantityObj") Quantity quantityObj,
                        Model model,
                        @ModelAttribute("value") Integer value)  {

    //we retrieve what user it is
    User user = userService.getUserByUsername(customUserDetailsService.returnUsername());
    CustomerProfile customerProfile;
    //if the user has no customer profile
    if(user.getCustomerProfile() == null){
        //redirect to profile
        return "profile";
    }else {
        //set this customer to his customerprofile
        customerProfile = user.getCustomerProfile();
    }

    //get the value of quantity from the quantityObj attribute value
    model.addAttribute( s: "value", value);

    //add the product to the database
    shoppingCartService.addProduct(value,id,customerProfile);

    //when added redirect to shop
    return "redirect:/shop";
}
```

fig2.4.5.1.4: CartController.java addToCart()

In viewProduct.html the form part of the code where the post method is defined. We can see the embedded Thymeleaf elements. The product id is specified: ${product.id}. A quantityObj where the field of value is required to be entered is also defined: required th:object="${quantityObj}" th:field="*{value}". The input value has a minimum value of 1 and max value of 10.

```
<form th:action="@{/addToCart/{id}(id = ${product.id})}" method="post">
    <div class="input-group row">
        <div class="col-xs-2">
            <label for="quantity" class="h3 py-2"><strong>Quantity:</strong></label>
            <input type="number" min="1" max="10" class="form-control "
                   required th:object="${quantityObj}" th:field="*{value}"
                   name="quantity" id="quantity">
        </div>
    </div>
```

fig2.4.5.1.5: viewProduct.html

The image(*fig2.4.5.1.6*) shows a registered customer selects 2 for the quantity on the viewProduct page. When the customer clicks the add to cart button the page is redirected to the shop page and the cart icon is update to reflect the number of items in the cart(*fig2.4.5.1.7*).

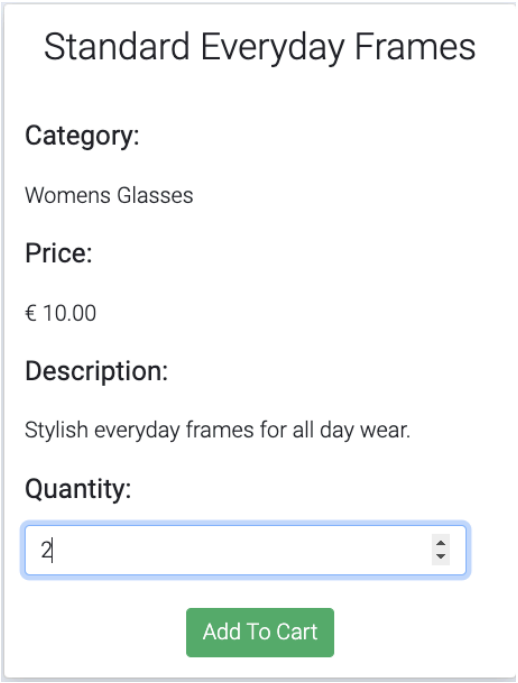

fig2.4.5.1.6: viewProduct.html the item description card.

fig2.5.4.1.7: shop.html the cart number is updated

The image(*fig2.4.5.1.8*) shows that a cartItems entry has been successfully inputted in the cart_items table. It records the information of the customer_profile_id, product_id and the quantity of the item that was added to cart.

	id	quantity	customer_profile_id	product_id
1	43	2	1	6

fig2.4.5.1.8: cart_items table in the database

2.4.6. Upload Prescription component.

This section describes how a registered customer is able to upload a prescription to the system. Once a customer has registered with the system ***2.4.3. Register Component*** and logged in ***2.4.4. Login Component*** they will have to complete a customer profile, that includes the fields of address, county, postcode, their left eye prescription, their right eye prescription and a digital file of their prescription in jpeg or png format. In the process of uploading prescription a customer must fill in a customerProfile, all the fields of the customerProfile must be entered before the customerProfile can be submitted. The image(*fig2.4.6.1*) shows the class diagram for the upload prescription component of the application. Again the UserService and the CustomUserDetailsService classes are the same service classes that have been described in ***2.4.3. Register Component*** and ***2.4.4. Login Component*** previously. They are included here in order to facilitate retrieving details of a logged in customer. There is also use of a CustomerProfileDTO class. The benefits of using a DTO class is that it can decouple persistence models from API models. DTO can be implemented so that we are only exposing a certain set of attributes of our entities to the front-end(*Cassiomolin, 2016*). The image(*fig2.4.6.2*) show the layered architectural view of the component.

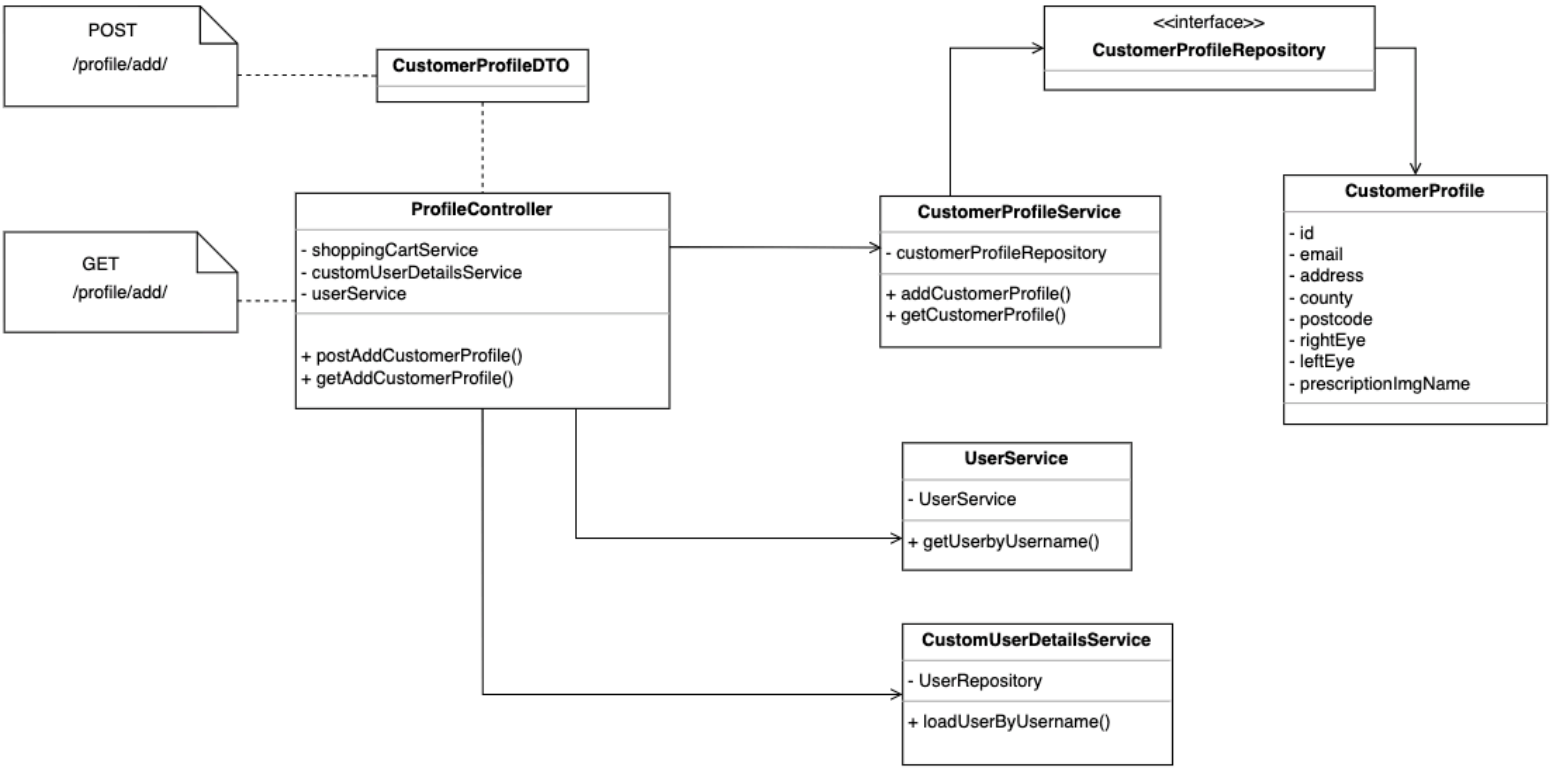

fig2.4.6.1: Class diagram for upload prescription component

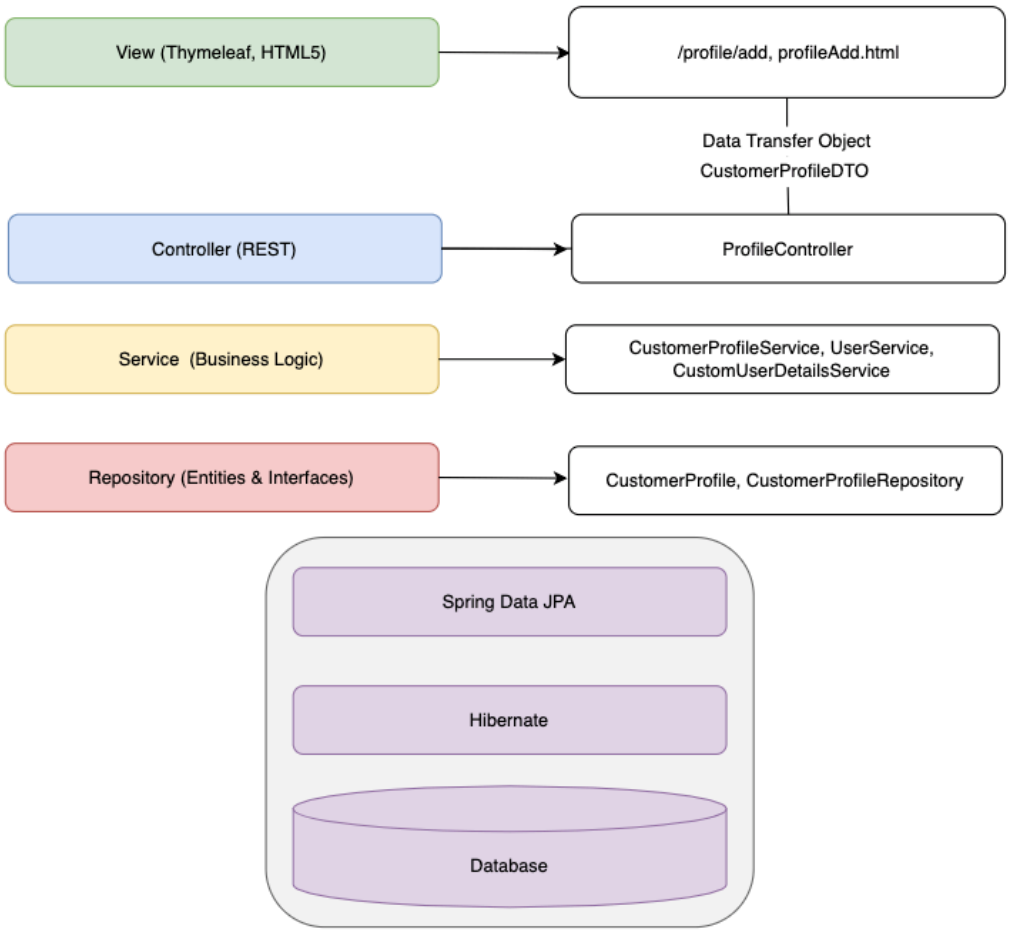

fig2.4.6.2: Architectural layered view of upload prescription component

2.4.6.1. Upload Prescription component code.

The image(*fig2.4.6.1.1*) shows the CustomerProfile class implementation. It has member variable attributes of id, email, address, county, postcode, rightEye, leftEye and the name of the prescription image file. Using the @Entity annotation this class will be mapped to a table in the database. The CustomerProfile has a one to one relationship with the User class. So in our database, the relationship is defined as one User can have one CustomerProfile.

```java
@Entity
@Table(name = "customer_profile")
public class CustomerProfile {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Integer id;

    @Column(name = "email", nullable = false)
    private String email;

    @Column(name = "address", nullable = false)
    private String address;

    @Column(name = "county", nullable = false)
    private String county;

    @Column(name = "postcode", nullable = false, length = 7)
    private String postcode;

    @Column(name = "right_eye", nullable = false)
    private double rightEye;

    @Column(name = "left_eye", nullable = false)
    private double leftEye;

    @Column(name = "prescription_img_name")
    private String prescriptionImgName;

    @OneToOne(fetch = FetchType.LAZY, cascade = CascadeType.ALL, mappedBy = "customerProfile")
    private User user;
```

fig2.4.6.1.1: CustomerProfile.java

The image(*fig2.4.6.1.2*) shows the CustomerProfileRepository interface that extends JpaRepository. JpaRepository has implementations of CRUD methods that we will use in the service layer.

```
@Repository
public interface CustomerProfileRepository extends JpaRepository<CustomerProfile, Integer> {
}
```

fig2.4.6.1.2: CustomerProfileRepository.java

The image(*fig2.4.6.1.3*) shows the CustomerProfileService class. This class has methods, addCustomerProfile(), that save the customerProfile to the database, a getCustomerProfile(), that finds the customerProfile by the id.

```
@Service
public class CustomerProfileService {

    @Autowired
    private CustomerProfileRepository customerProfileRepository;

    public void addCustomerProfile(CustomerProfile customerProfile){
        customerProfileRepository.save(customerProfile);
    }

    public Optional<CustomerProfile> getCustomerProfile(Integer id){
        return customerProfileRepository.findById(id);
    }

}
```

fig2.4.6.1.3: CustomerProfileService.java

In the ProfileController GET and a POST API methods have been defined for the /profile/add/ endpoints. The image(*fig2.4.6.1.4*) shows the GET method, getAddCustomerProfile(). In the method body attributes of the user, lastName, firstName, email are set to the model. A new CustomerProfileDTO object is set in the model. Cart information which is depicted in (*fig2.4.6.1.7*) is also set in the model with the variable cartCount.

```
@GetMapping("/profile/add")
public String getAddCustomerProfile(Model model){
    //get the current user
    User user = userService.getUserByUsername(customUserDetailsService.returnUsername());
    //output the existing details
    model.addAttribute( s: "lastName", customUserDetailsService.returnLastName());
    model.addAttribute( s: "firstName", customUserDetailsService.returnFirstName());
    model.addAttribute( s: "email", customUserDetailsService.returnUsername());

    //the data transfer object will hold the attrbutes of the CustomerProfile
    model.addAttribute( s: "customerProfileDTO", new CustomerProfileDTO());

    // if user already has a CustomerProfile, display the cart number
    if(user.getCustomerProfile() != null){
        CustomerProfile customerProfile = customerProfileService.getCustomerProfile(
                                          user.getCustomerProfile().getId()).get();
        Integer totalQuantity = shoppingCartService.cartCount(
                                          shoppingCartService.listCartItems(customerProfile));
        model.addAttribute( s: "cartCount", totalQuantity);
    }
    //display the profile add page
    return "profileAdd";
}
```

fig2.4.6.1.4: ProfileController.java getAddCustomerProfile()

The image(*fig2.4.6.1.5*) shows postAddCustomerProfile the POST API method for the REST end-point /profile/add/. In the method parameters we have access to customerProfileDTO object, a multipartFile variable for uploading a file to the system, and the variable for the prescriptionImgName. These are received from the model and will be processed in the method body. First we find if the user has a CustomerProfile yet. If they have we set a CustomerProfile object with CustomerProfile they have previously entered. If it doesn't exist, we just initialise a

new CustomerProfile object. After this we copy the attributes from our DTO object Customer-ProfileDTO to the CustomerProfile object. The attributes are email, address, county, postcode, left eye and right eye prescription.

```
@PostMapping("/profile/add")
public String postAddCustomerProfile(@ModelAttribute("customerProfileDTO") CustomerProfileDTO customerProfileDTO,
                                     @RequestParam("prescriptionImage") MultipartFile multipartFile,
                                     @RequestParam("imgName") String prescriptionImgName) throws IOException{

    // get the user
    User user = userService.getUserByUsername(customUserDetailsService.returnUsername());
    CustomerProfile customerProfile;
    //if customerProfile exists then assign this user to this customerProfile
    if(user.getCustomerProfile() != null){
        Integer customerId = user.getCustomerProfile().getId();
        customerProfile = customerProfileService.getCustomerProfile(customerId).get();
    }else{
        //initalise the customerProfile
        customerProfile = new CustomerProfile();
        //set and id
        customerProfile.setId(customerProfileDTO.getId());
    }
    //set all the attributes
    customerProfile.setEmail(user.getEmail());
    customerProfile.setAddress(customerProfileDTO.getAddress());
    customerProfile.setCounty(customerProfileDTO.getCounty());
    customerProfile.setPostcode(customerProfileDTO.getPostcode());
    customerProfile.setRightEye(customerProfileDTO.getRightEye());
    customerProfile.setLeftEye(customerProfileDTO.getLeftEye());
```

fig2.4.6.1.5: ProfileController.java getAddCustomerProfile()

The image(*fig2.4.6.1.6*) shows the code for uploading the prescription image to the system. If the multipartFile is not empty set the name of the file to its filename, set the Path variable where the file is to be uploaded, “uploadDir”(*fig2.4.6.1.7*) and with “imageIdentifier”, then write the file to the path variable in image. If the multipartFile is empty just assign the “imageIdentifier” the empty string value and set the customerProfile’s setPrescriptionImgName that value.

```
//setting a variable for the name
String imageIndenifier;

//if the multipartFile object is not empty in passing from the front-end
if(!multipartFile.isEmpty()){
    //assign the filename
    imageIndenifier = multipartFile.getOriginalFilename();
    // this creates the path where the image is to be uploaded to the system
    Path fileNameAndPath = Paths.get(uploadDir, imageIndenifier);
    //this writes the actual file and uploads to the system
    Files.write(fileNameAndPath, multipartFile.getBytes());
}else {
    //the file is empty
    imageIndenifier = prescriptionImgName;
}
customerProfile.setPrescriptionImgName(imageIndenifier);
```

fig2.4.6.1.6: ProfileController.java getAddCustomerProfile() continued

```
// setting a variable for the directory of the product images
public static String uploadDir = System.getProperty("user.dir") + "/src/main/resources/static/prescriptionImages";
```

fig2.4.6.1.7: ProfileController.java path declaration private member variable

The image(*fig2.4.6.1.8*) shows the customerProfile and user information being saved to the system. The customerProfile is set and the users relation to that customerProfile is saved too. Once this is completed the system redirects back to the profile page.

```
//this writes to the db
customerProfileService.addCustomerProfile(customerProfile);
//this sets the foreign key on the Users table linking the new entry
user.setCustomerProfile(customerProfile);
//set the CustomerProfile as the this user
customerProfile.setUser(user);
//save the suer
userService.saveUser(user);
//redirect back to the profile
return "redirect:/profile";
```

fig2.4.6.1.8: ProfileController.java getAddCustomerProfile() continued

In profileAdd.html(*fig2.4.6.1.9*), bootstraps inbuilt class for opening the file browser that allows us to upload a file is included “custom-file-input”. The image is displayed when the file is loaded. When submitted, all details of the customerProfile are sent in the customerProfileDTO object using the POST method.

```
<p><strong>Prescription Image</strong></p>
<div class="custom-file">
    <!-- we use the variable name prescriptionImage here as the passed
    object to the controller-->
    <input type="file" class="custom-file-input" name="prescriptionImage"
           accept="image/jpeg, image/png, image/pdf"  id="prescriptionImage"/>
    <label class="custom-file-label" for="prescriptionImage">Choose file</label>
</div>
<div class="form-group">
    <img src="#" id="imgPreview" height="100px" width="100px"
         th:src="@{${'/prescriptionImages/' + customerProfileDTO.prescriptionImgName}}"
         style="margin-top: 20px" alt=" ">
</div>
<input type="hidden" name="imgName" th:value="${customerProfileDTO.prescriptionImgName}">
<button type="submit" class="btn btn-success">Submit</button>
```

fig2.4.6.1.9: ProfileAdd.html upload prescription section of the code.

After registering and logging into the system. The system will require the customer to make a customerProfile by clicking the add more details to your profile button(*fig2.4.6.1.10*).

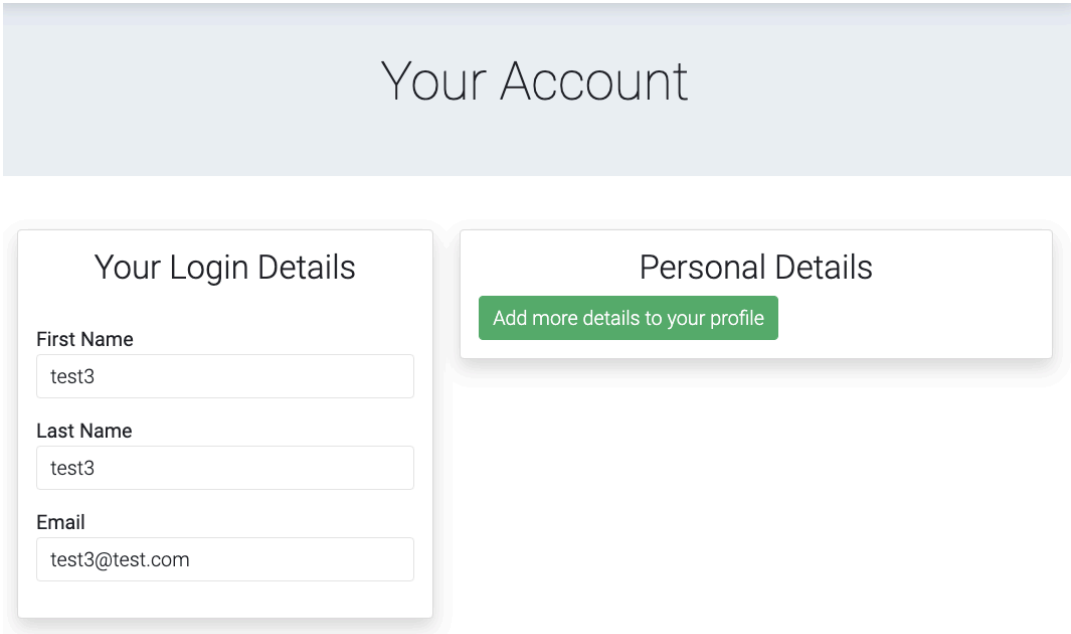

fig2.4.6.1.10: Profile.html upload prescription section of the code.

The image shows the page(*fig2.4.6.1.11*) where the customer has to fill out their details including prescription for left-eye, right-eye and upload a prescription image.

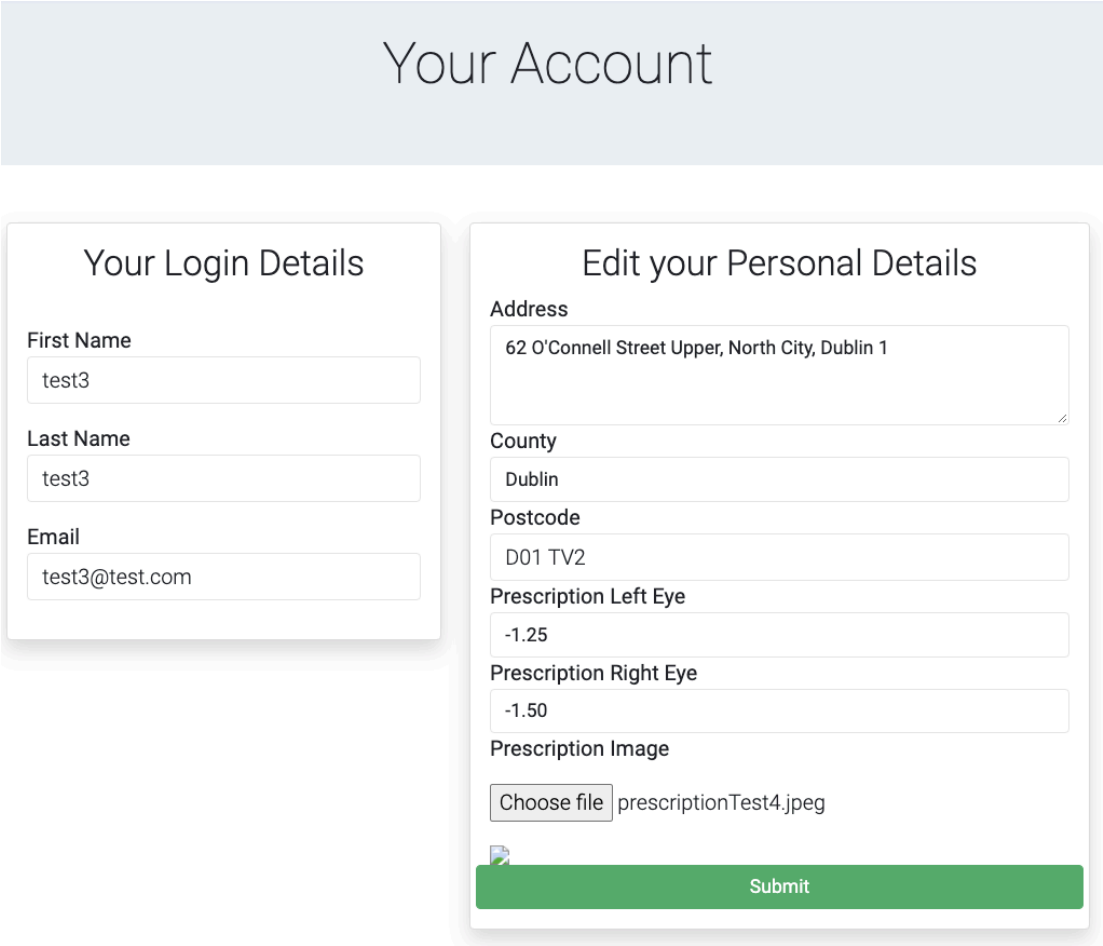

fig2.4.6.1.11: ProfileAdd.html upload prescription section of the code.

The image(*fig2.4.6.1.12*) shows after the customerProfile is submitted they are redirected back to the profile page with updated details.

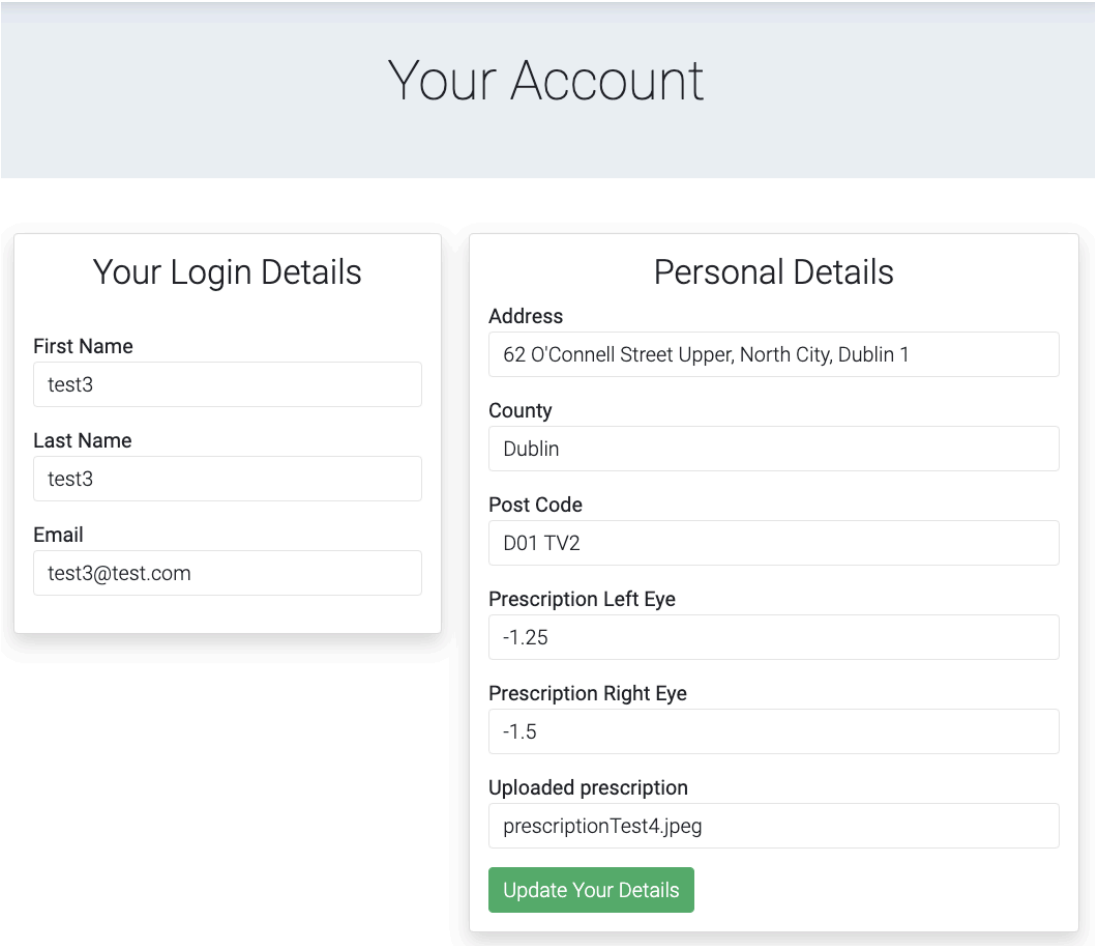

fig2.4.6.1.12: Profile.html customer details added.

The image(*fig2.4.6.1.13*) shows that the customer_profile table has been updated with a new entry for test3 customer. All the attributes have been written successfully.

	id	address	county	email	lef...	po...	prescription_img...	right_eye
1	1	190 The New Hardwicke , Chu…	Dublin	test@test.com	-2.25	D07 XY4	test prescription.jpeg	-4.25
2	37	Mayor Street Lower, Interna…	Dublin	test1@test.com	-1.75	D07 XY4	prescriptiontest2.jpeg	-2
3	38	Dublin City University, Col…	Dublin	test2@test.com	-2	D07 XY4	prescriptionTest3.jpeg	-1.75
4	40	62 O'Connell Street Upper, …	Dublin	test3@test.com	-1.25	D01 TV2	prescriptionTest4.jpeg	-1.5

fig2.4.6.1.13: customer_profile table with added test3 customer profile.

The image shows the user table where the field for customer_profile is also updated with the primary key id of the customer_profile corresponding to that user test3. The id reference is 40 for user test3.

	user_id	email	first_name	last_name	pass...	customer_profile_id
1	1	christobelchan@gmail.com	chris	chan	$2a$10$AVWm…	<null>
2	2	test@test.com	test	test	$2a$10$CpO5…	1
3	17	test1@test.com	test1	test1	$2a$10$2qUP…	37
4	18	test2@test.com	test2	test2	$2a$10$7OTi…	38
5	39	test3@test.com	test3	test3	$2a$10$RTjl…	40
6	41	test5@test.com	test5	test5	$2a$10$3PJq…	<null>

fig2.4.6.1.14: user table with added test3 customer_profile_id updated value

2.4.7. Make Payment component.

The make payment component of the application allows a registered customer that has added items to a cart and checked out, to pay for those items. The Stripe API was used to implement secure payment of goods using credit/debit cards. The web application will have a custom page for payments and Stripe API will integrate sending card details securely to Stripe servers for processing. The process flow is described in the image(*fig2.4.7.1*) was taken from Stripe.

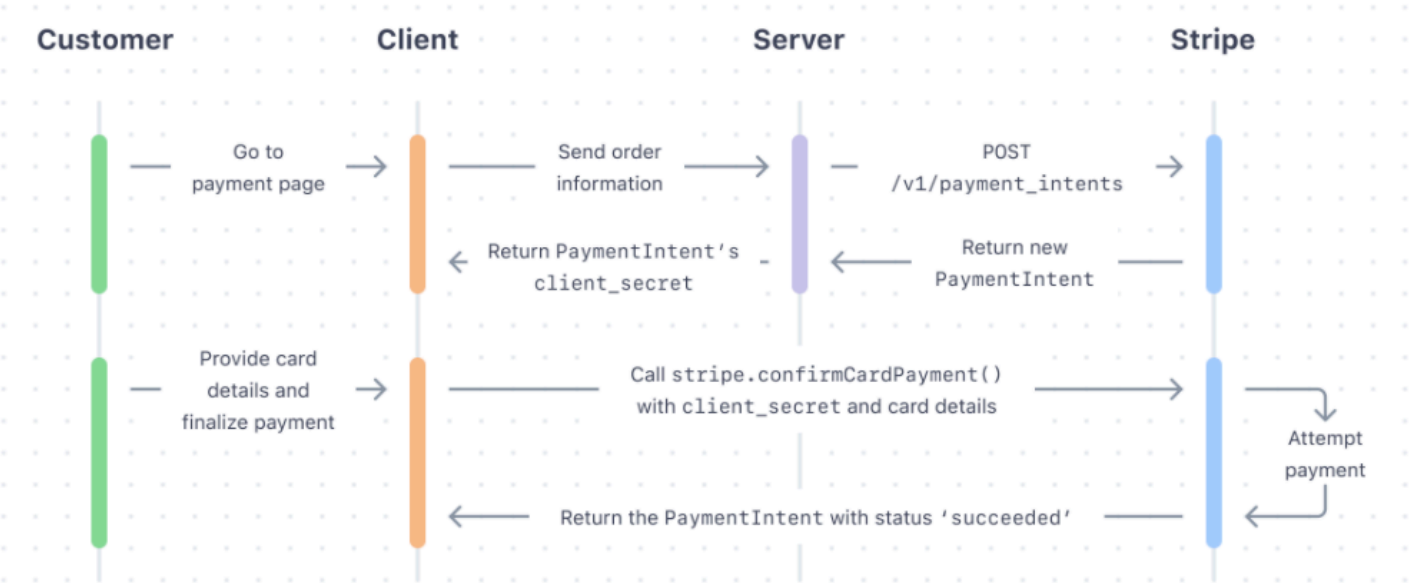

fig2.4.7.1: sequence diagram for stripe payment

Steps involved in the transaction:

1. **Create a PaymentIntent:** The paymentIntent is an object that is used to collect payment from the customer. In our application the PaymentIntent object contains the amount of money being transacted and the currency in which the transaction is taking place. The PaymentIntent is sent to Stripe servers. We need our Stripe public and private keys to make the PaymentIntent, they must be declared at this stage.
2. **Secret Client Id returned:** The returned PaymentIntent now holds a client secret id. Step 1 and step 2 are performed in sequence when we load the payment.html in our application.
3. **Customer Sends Details:** The customer enters their card details and pushes the pay button on the payment.html page. A script will run that will executes a confirmCardPayment() method with the client secret id and card details, sending it to Stripe servers.
4. **Transaction Result:** Then Stripe sends back the result of the PaymentIntent whether it succeeded or failed.

2.4.7.1. Make Payment component code.

The CartController is where the payment.html(*fig2.4.7.1.1*) end-point is defined. We set a number of objects to be rendered on the page including the email of the customer, the total cost of the cartItems and our Stripe public key which is needed to create the paymentIntent.

```
@GetMapping("/payment")
public String home(Model model){
    //get the user
    User user = userService.getUserByUsername(customUserDetailsService.returnUsername());
    //get the customer Profile
    CustomerProfile customerProfile = user.getCustomerProfile();
    //get the cartItems of the customer Profile
    List<CartItem> cartItemList = shoppingCartService.listCartItems(customerProfile);
    DecimalFormat decimalFormat = new DecimalFormat( pattern: "0.00");

    //set the objects to the model and load the page
    model.addAttribute( s: "email", customUserDetailsService.returnUsername());
    model.addAttribute( s: "total", decimalFormat.format(shoppingCartService.totalCost(cartItemList)));
    model.addAttribute( s: "stripePublicKey", stripePublicKey);
    return "payment";
}
```

fig2.4.7.1.1: CartController.java mapping for payment.html

Our public key is set in the properties file at: */src/main/resources/applications.properties.* (*fig2.4.7.1.2*) The value is then injected to CartController.java as a private member variable in the image(*fig2.4.7.1.3*). The Stripe private key is initialised on the system when the application starts(*fig2.4.7.1.4*) in the file at: */src/main/java/SimplefocalsApplication.java*

```
#this is the public key for the stripe api
stripe.public.key=pk_test_51JHr20IkXPmyXB6SiaFnnf7luD1gsib5Qgfk7GkUiQ1ISPElruFtN39LzmrQlIT9fEZvlxmaA48llJcwyK2OQN4O00cjSuST6I
```

fig2.4.7.1.2: applications.properties public stripe key

```
@Value("${stripe.public.key}")
private String stripePublicKey;
```

fig2.4.7.1.3: CartController.java public stripe key injection

```
@SpringBootApplication
public class SimplefocalsApplication {

    //this is the private key for my Stripe account.
    @PostConstruct
    public void setup(){
        Stripe.apiKey="sk_test_51JHr20IkXPmyXB6SMX8ekXNvL2Zf4QOJGWP16Kjbwsx" +
                "7XVIAnbpH3rhr1EdCZkLmuVTs1tsU30KgdNDG4IhZQ0fz0053QCUEwC";
    }
```

fig2.4.7.1.4: SimplefocalsApplication.java private stripe key Initialised

From Stripes developer section of the website, we include the given html file for formatting of the payment.html, css file for styling of the payment.html and Javascript file for processing of the payment. Analysing the main parts of the client.js file from Stripe. The paymentIntent de-

scribed in ***2.5.6. Make Payment component. Steps involved in transaction - 1***, is shown in the image(*fig2.4.7.1.5*).

```
document.querySelector( selectors: "button").disabled = true;
fetch( input: "/create-payment-intent",  init: {
    method: "POST",
    headers: {
        "Content-Type": "application/json"
    },
    body: JSON.stringify(purchase)
```

fig2.4.7.1.5: client.js paymentIntent

The paymentIntent described in ***2.5.6. Make Payment component. Steps involved in transaction - 3*** confirmCardPayment(*fig2.4.7.1.6*), is shown in the image.

```
var payWithCard = function(stripe, card, clientSecret) {
    loading( isLoading: true);
    stripe
        .confirmCardPayment(clientSecret, {
            receipt_email: document.getElementById( elementId: 'email').value,
            payment_method: {
                card: card
            }
        })
```

fig2.4.7.1.6: client.js confirmCardPayment()

The CreatePaymentIntent post mapping (*fig2.4.7.1.7*) is found in the StripePaymentController.-java. In the method body we get details of the purchase, the customerProfile, and the total amount about to be transacted. The attributes are passed to a PaymentIntent builder object and then the paymentIntent can be created. Once the paymentIntent is created we can move the items to OrderHistory and empty the CartItems table.

```java
@PostMapping("/create-payment-intent")
public CreatePaymentResponse createPaymentIntent() throws StripeException {

    //get the user details
    User user = userService.getUserByUsername(customUserDetailsService.returnUsername());
    //get the customerProfile
    CustomerProfile customerProfile = user.getCustomerProfile();
    //get the cartItems of that customerProfile
    List<CartItem> cartItemList = shoppingCartService.listCartItems(customerProfile);
    //get the total cost
    long totalCost = (long)shoppingCartService.totalCost(cartItemList);

    //set the PaymentIntent parameters with builder object,
    // the currency euro and the total amount tendered
    PaymentIntentCreateParams createParams = new PaymentIntentCreateParams.Builder()
            .setCurrency("eur")
            .setAmount(totalCost*100L)
            .build();
    // Create a PaymentIntent with the order amount and currency
    PaymentIntent intent = PaymentIntent.create(createParams);
    //delete cartItems
    moveCartItemsToOrderHistory(cartItemList);
    //create the
    return new CreatePaymentResponse(intent.getClientSecret());
}
```

fig2.4.7.1.7: StripePaymentController CreatePaymentResponse()

Let's examine the flow of actions the user goes through to make a purchase. From the check-out page the user selects pay now(*fig2.4.7.1.8*).

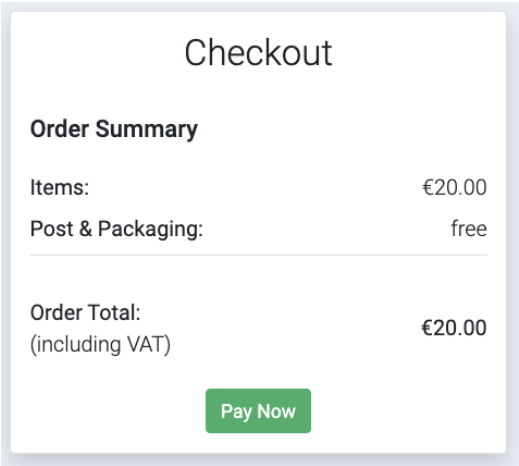

fig2.4.7.1.8: checkout.html user checks out

The system loads the payment page(*fig2.4.7.1.9*), in this example the user with email test@test.com will pay 20 euros for the items. They have to enter a valid card number and expiry date and card cvc. Since this is a test application stripe provides test card numbers that can be used. We shall use the default visa test number 4242 4242 4242 4242, the expiry date and card cvc can be any values(*fig2.4.7.1.10*).

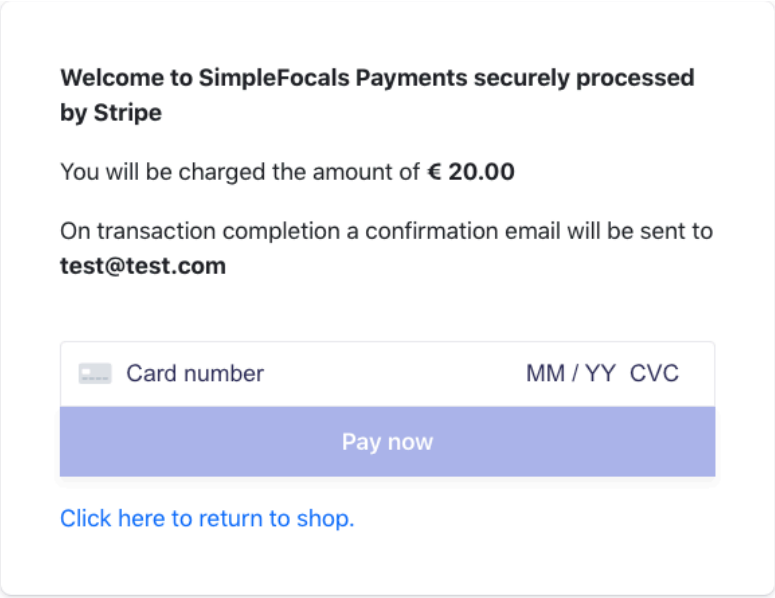

fig2.4.7.1.9: payment.html page

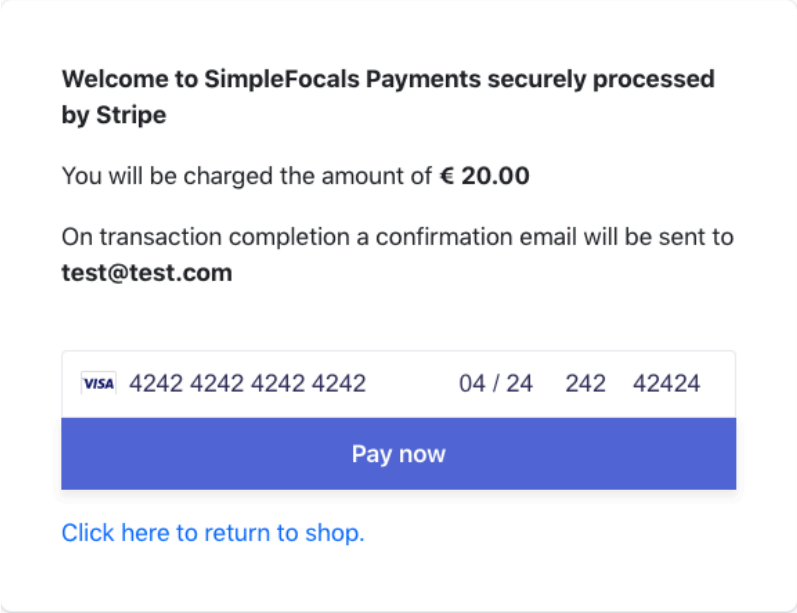

fig2.4.7.1.10: payment.html page with card details

When the test user presses the Pay now button the confirmCardPayment() will be called and process the transaction. We can see the transaction was a success(*fig2.4.7.1.11*). We can view details in our stripe dashboard(*fig2.4.7.1.12*). The user can click on the link to return to shop.

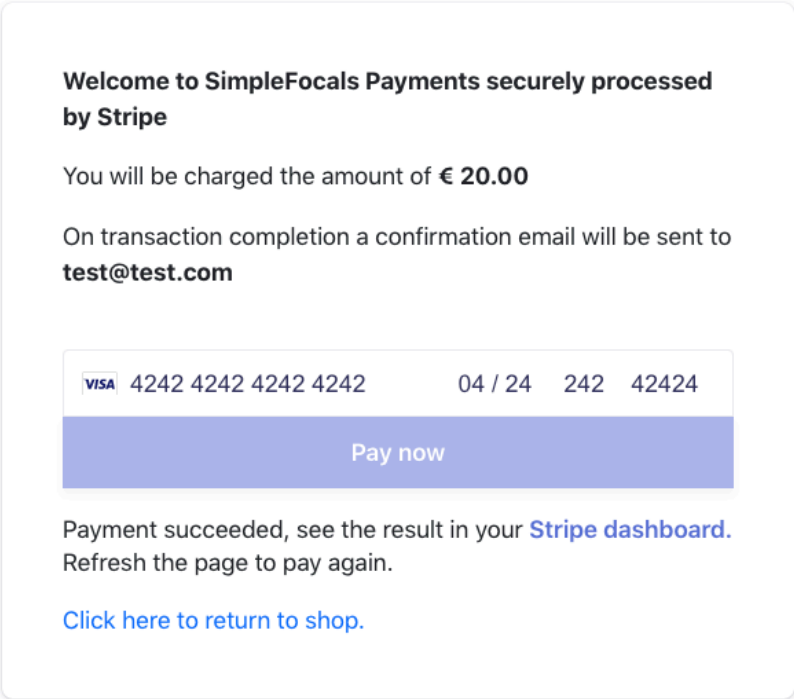

fig2.4.7.1.11: payment.html successful transaction

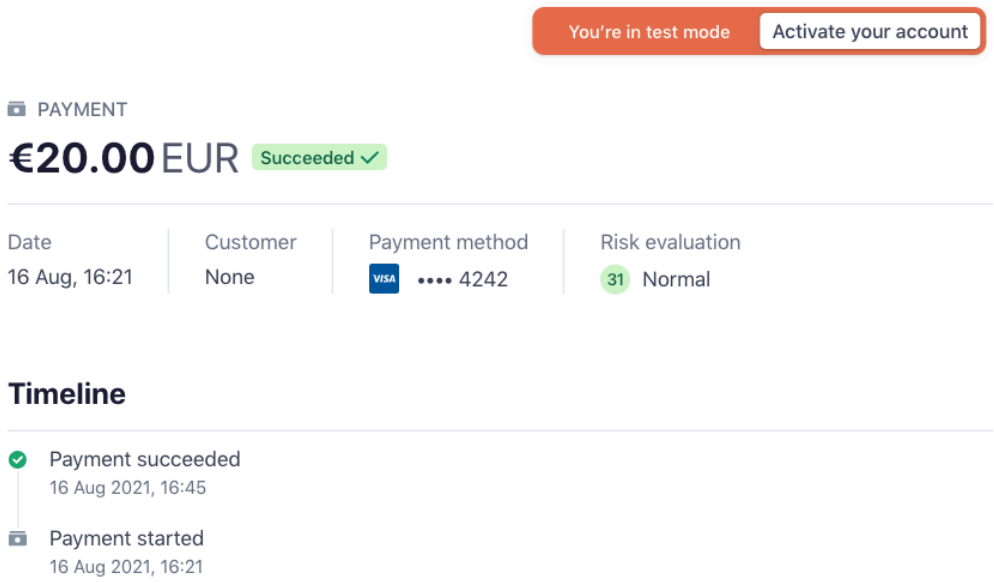

fig2.4.7.1.12: Stripe Dashboard confirming the payment

The order_history(*fig2.4.7.1.13*) table then adds a record of the transaction.

	id	customer_id	date_created	product_id	quantity	subtotal
1	33	1	today	6	1	10
2	48	1	today	6	2	20

fig2.4.7.1.13: order_history table updated

2.4.8. Add Product Component

The add product component describes the function of a logged in administrator adding a product to the system. The administrator will click the manage button on the products section. From the Products page they shall select the add product button. The administrator shall fill out all the attributes of the product including name, category, price product description and select the product image. When the administrator presses the submit button the product is added to the system. Now the product is viewable in the shop section of web application. The image(*fig2.4.8.1*) shows the class diagram for the add product component.

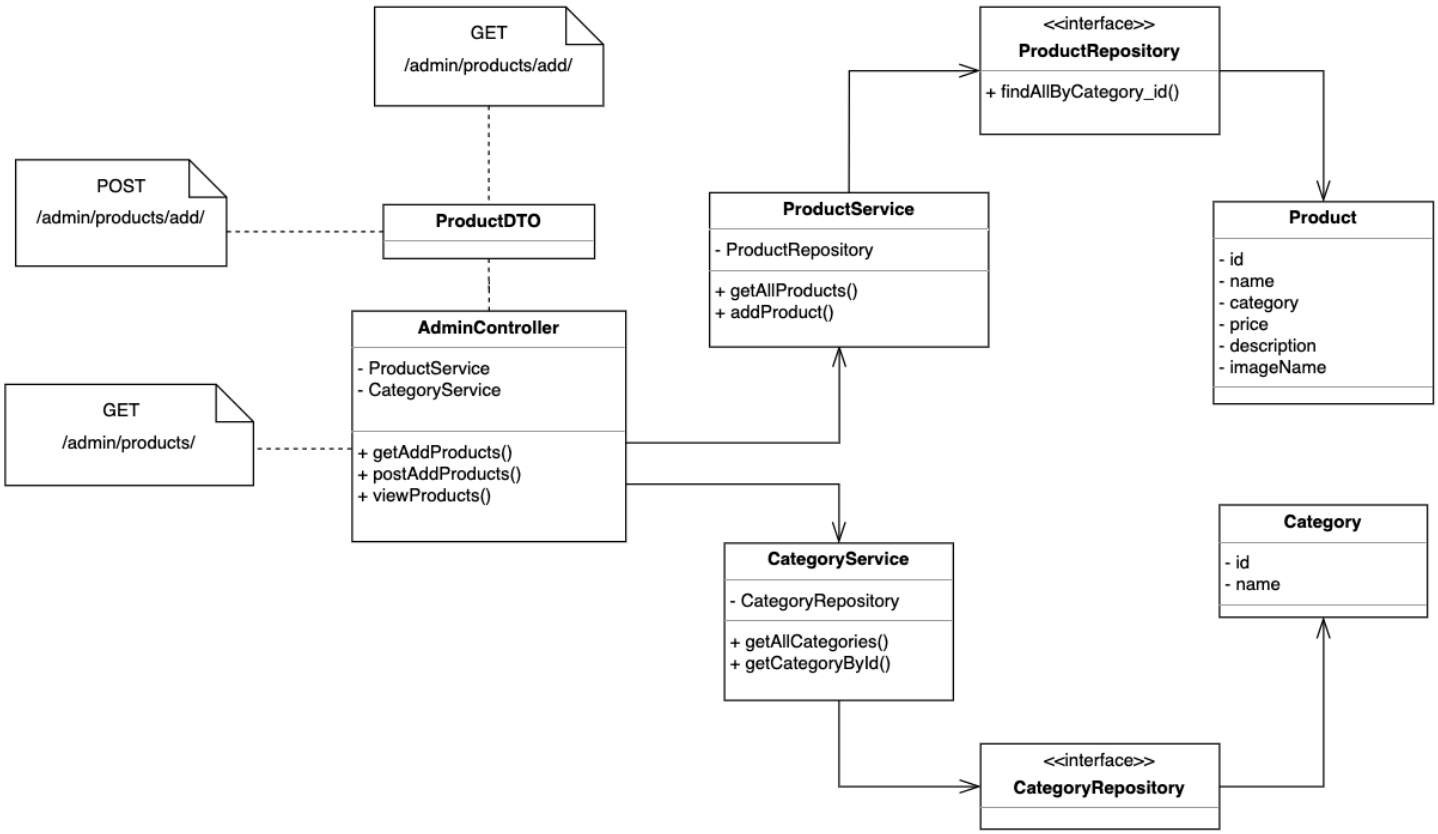

fig2.4.8.1: class diagram for the add product component

The image(*fig2.4.8.1*) shows the layered architectural view of the add product component of the system.

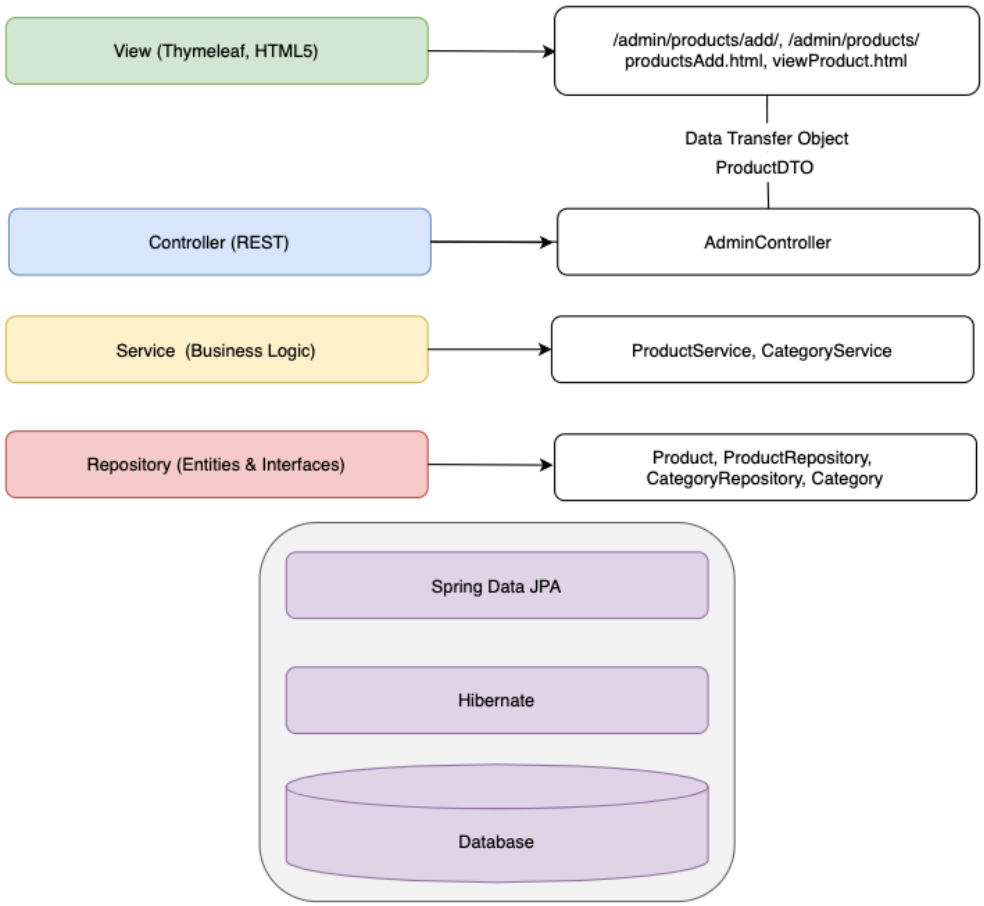

fig2.4.8.1: The add product component in layered system architecture view

2.4.8.1. Add Product Component code

Similar to ***2.4.2. View Product Component*** in this document, we use the Product and Category entity classes for this component of the application. The ProductRepository and Category-Repository classes both extend JpaRepository and provide CRUD functionality in modifying product and category tables in the service layer of the application. The CategoryService has methods we will use to retrieve the category of the product. As defined previously the relation-ship between product and category is: many products can belong to one category and one category can have many products. The image shows the ProductService class, addProduct() is the method used to save a product to the database.

```java
@Service
public class ProductService {

    @Autowired
    private ProductRepository productRepository;

    public List<Product> getAllProducts() { return productRepository.findAll(); }

    public void addProduct(Product product) { productRepository.save(product); }
```

fig2.4.8.1.1: The ProductService class with addProduct() definition.

The AdminController.java is the class that has the REST API methods that facilitate the add product functionality. The image(*fig2.4.8.1.2*) shows the viewProducts() in the administrators CMS. A list of products is set to the model to be rendered by Thymeleaf on the products.html.

```java
@GetMapping("/admin/products")
public String viewProducts(Model model){
    List<Product> products = productService.getAllProducts();
    model.addAttribute( s: "products", products);
    return "products";
}
```

fig2.4.8.1.2: AdminController.java class viewProducts()

The image(*fig2.4.8.1.3*) shows the getAddProducts() method in the AdminController. This method is the GET method that will be called when the productAdd.html is loaded. A product-DTO object and categories object will be embedded sent to the model by Thymeleaf.

```java
@GetMapping("/admin/products/add")
public String getAddProducts(Model model){
    model.addAttribute( s: "productDTO", new ProductDTO());
    model.addAttribute( s: "categories", categoryService.getAllCategories());

    return "productsAdd";
}
```

fig2.4.8.1.3: AdminController.java class getAddProducts()

The image(*fig2.4.8.1.4*) shows the post method that adds the product to the system. This is a POST method that is called when the administrator selects the submit button on the products-Add.html after entered all the product details. Again similar to ***2.4.6. Upload Prescription component*** in the method, the parameters takes a ProductDTO object, a multipartFile variable for uploading a file to the system, and the variable for the imgName. These objects are received from the model. A new Product object is created and the productDTO is used to transfer the attribute values to the new Product. Fields of the id, the product name, the category, the price and the description are set in the Product.

```
@PostMapping("/admin/products/add")
public String postAddProducts(@ModelAttribute("productDTO") ProductDTO productDTO,
                              @RequestParam("productImage") MultipartFile multipartFile,
                              @RequestParam("imgName") String imageName) throws IOException {

    Use the productDTO obj to transfer the attributes to product object
    Product product = new Product();
    product.setId(productDTO.getId());
    product.setName(productDTO.getName());

      we need to pass the category to the product
      the ProductDTO only has the category id
    Integer categoryId = productDTO.getCategoryId();
    String categoryName = productDTO.getName();

    // set the attributes for the product
    product.setCategory(categoryService.getCategoryById(productDTO.getCategoryId()).get());
    product.setPrice(productDTO.getPrice());
    product.setDescription(productDTO.getDescription());
```

fig2.4.8.1.4: AdminController.java class postAddProducts()

Similar to the ***2.4.6. Upload Prescription component*.** when uploading the prescription image, a multiPartFile was used to upload the product image to the system(*fig2.4.8.1.5*). The upload

directory(*fig2.4.8.1.6*) of the file and the file name have to passed to Path variable before being written to the system. Once the files is written to the specific path, the imageIdentifier is then set as the ImageName of the product before being saved to the database.

```
//Name of the image
String imageIdentifier;
//if the multipartFile object is not empty in passing from the front-end
if(!multipartFile.isEmpty()){
    imageIdentifier = multipartFile.getOriginalFilename();
    // this creates the path where the image is to be uploaded to the system
    Path fileNameAndPath = Paths.get(uploadDir, imageIdentifier);
    //this writes the actual file and uploads to the system
    Files.write(fileNameAndPath, multipartFile.getBytes());
}else {
    //the file is empty
    imageIdentifier = imageName;
}
//set the last attributes of the product
product.setImageName(imageIdentifier);
//save the object to db
productService.addProduct(product);

//redirect to products page
return "redirect:/admin/products";
```

fig2.4.8.1.5: AdminController.java class postAddProducts() continued

```
// setting a variable for the directory of the product images
public static String uploadDir = System.getProperty("user.dir")
        + "/src/main/resources/static/productImages";
```

fig2.4.8.1.6: AdminController.java path for the product image

In the productsAdd.html(*fig2.4.8.1.7*), the Post method type is defined with the REST API end-point /admin/products/add/ using Thymeleaf. A productDTO object is used in the form to take the inputs from the form and pass it to the controller for processing. The enctype="multipart/form-data" outlines how the data should be encoded when submitting through the form.

```
<form th:action="@{/admin/products/add}" method="post"
      th:object="${productDTO}" enctype="multipart/form-data">
```

fig2.4.8.1.7: productsAdd.html form tag

The image(*fig2.4.8.1.8*) show the section of html using bootstrap class to have file input from the browser. This is used to upload the product image to the system. Using the custom-file class facilitates this action, the input can be set to file and the class custom-file-input is called.

```
<div class="custom-file">
    <!--use the custom-file-input to upload the file-->
    <input type="file"  class="custom-file-input" name="productImage"
           accept="image/jpeg, image/png"  id="productImage"/>
</div>
```

fig2.4.8.1.8: productsAdd.html file input

Let's see how this component is used in the application. When the administrator navigates to the add product section of the application. They will fill out all the fields of the product(*fig2.4.8.1.9*). When they push submit the page is redirected to the products page. The administrator can see the product has been added successfully(*fig2.4.8.1.10*).

Add a new Product

Name

Gold Trim Sunglasses

Select Category

Mens Glasses

Price

15

Product Description

These are stylish gold frame sunglasses perfect for any sunny day.

Product Image

Choose file gold sunglasses

Submit

fig2.4.8.1.9: productsAdd.html administrator enters a new product

Product Section

Add Product

No	Name	Description	Category	Image	Delete
1	Everyday Basic Frames	These comfortable frames will easily fit into your daily life.	Mens Glasses		Delete
2	Standard Everyday Frames	Stylish everyday frames for all day wear.	Womens Glasses		Delete
3	Everyday Rectangle Frames	Versatile everyday frames for a smart look.	Womens Glasses		Delete
4	Gold Trim Sunglasses	These are stylish gold frame sunglasses perfect for any sunny day.	Mens Glasses		Delete

fig2.4.8.1.10: products.html administrator enters a new product

When the product is added to the system the database is updated with the entry(*fig2.4.8.1.11*) and the product image is copied to the productImages directory(*fig2.4.8.1.12*).

id	description	image_name	name	price	category_id
3	These comfortable frames will easily fit in…	standard basic…	Everyday Basic Frames	10	2
6	Stylish everyday frames for all day wear.	standard women…	Standard Everyday Frames	10	4
50	Versatile everyday frames for a smart look.	standard recta…	Everyday Rectangle Fram…	10	4
51	These are stylish gold frame sunglasses per…	gold sunglasse…	Gold Trim Sunglasses	15	2

fig2.4.8.1.11: product table in the db with new entry

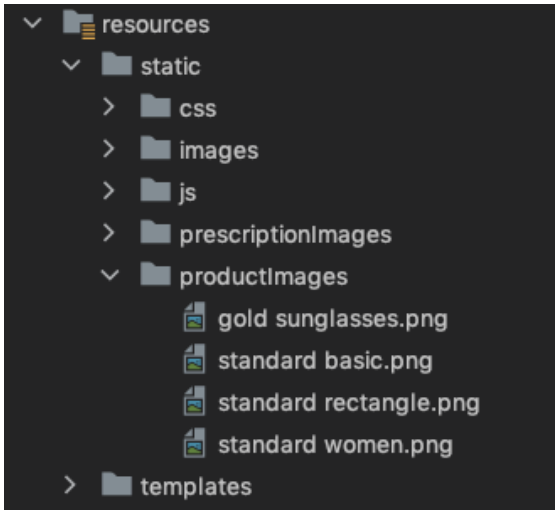

fig2.4.8.1.12: productImages directory

2.4.9. Extra Components coded.

This section describes other functionalities that were implemented outside the scope of this project. The core concept of this project was to develop a functioning e-commerce store that facilitates selling SimpleFocals range of prescription eyewear. In the process of implementing the functional requirements that were outlined in ***Requirements Specification Document,***

these functionalities were also coded in the development phase. Specifications for these could be detailed further in the next revision for this project.

- Cart Items number consistently displayed on all pages of the application(except payment page) for customers that have logged in and filled out customer profile details.
- When a customer has registered and logs in for the first time. They must fill out a Customer Profile, clicking on any link redirect to the add profile page.
- Add Category function for the CMS.
- Delete product function for the CMS.
- Delete Category function for the CMS.
- Registered customer who has made a customer profile has the ability to update their details.
- Checkout page implemented
- Remove Items from the Cart
- Items in the Cart are remembered for each customer on log in and out.

2.5. Testing

This section describes some of the testing that was carried out during development. The application was tested using a combination of manual testing of the components (GUI testing) and code based tests cases(to test method functionality). All coded test cases are located at the path: */src/test/java/com.github.simplefocals/*

2.5.1. View Product Component

This section describes one test case for view product function of the application. A test case for findAllByCategory_Id(), a method that retrieves all products for a given product Id, was written. This method is defined in the ProductRepository class we want to test if the function retrieves the correct information from the database. This method is used for filtering products in the shop when a customer clicks on a certain category of products to view. The images(*fig2.5.1.1*) (*fig2.5.1.2*)shows 4 products in the product table and 2 categories in the category table. The product table has a foreign key of category_id as an attribute.

	id	description	image_name	name	price	category_id
1	3	These comfortable frames will eas…	standard basic…	Everyday Basic Frames	10	2
2	6	Stylish everyday frames for all d…	standard women…	Standard Everyday Frames	10	4
3	50	Versatile everyday frames for a s…	standard recta…	Everyday Rectangle Fram…	10	4
4	51	These are stylish gold frame sung…	gold sunglasse…	Gold Trim Sunglasses	15	2

fig2.5.1.1: product table

	category_id	name
1	2	Mens Glasses
2	4	Womens Glasses

fig2.5.1.2: category table

The test case implementation is shown in the image(fig2.5.1.3). We are trying to find all the products that correspond to the category_id = 4. After retrieving them, they shall be printed to console and that the expected result of the test is: number of products with category_id = 4 is 2 items.

```java
@Autowired
private ProductRepository productRepository;

@Test
public void testAllProductsWithCategoryId(){

    //find all by the categoryId = 4, category womens glasses
    List<Product> productsByCategory = productRepository.findAllByCategory_Id(4);

    for (Product product: productsByCategory) {
      System.out.println(product.toString());
    }
    //there is only two items for womens glasses
    assertThat(productsByCategory.size()).isEqualTo(2);
}
```

fig2.5.1.3: ViewProductTest.java testAllProductsWithCategoryId()

From the image(*fig2.5.1.4*) it can be seen that the test was passed. The products that have a category_id = 4, were printed to console(*fig2.5.1.5*). The test behaved in the way that was expected

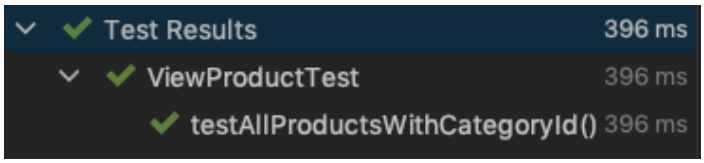

fig2.5.1.4: successful test result

```
Product{id=6, name='Standard Everyday Frames', c
Product{id=50, name='Everyday Rectangle Frames',
```

fig2.5.1.5: products with category_id = 4 printed to console

2.5.2. Register Component

This section describes one test case of the register component of the application. When a customer registers on the system. They should supply the fields of first name, last name, email and password. Of these fields the email address should be unique. If it is not the system should inform the user to enter a different email address as the email has been previously registered. The image(*fig2.5.1.1*) shows the user table in the database. This test case will try and register a test@test.com email(*fig2.5.1.2*) as a new user even though it exists in the system already. The expected result is that the system rejects the email and tells the user to register with different email.

	user_id	email	first_name	last_name	password	customer_profile_id
1	1	christobelchan@gmail.com	chris	chan	$2a$10$AVWm/…	<null>
2	2	test@test.com	test	test	$2a$10$Cp05h…	1
3	17	test1@test.com	test1	test1	$2a$10$2qUPA…	37
4	18	test2@test.com	test2	test2	$2a$10$7OTik…	38
5	39	test3@test.com	test3	test3	$2a$10$RTjld…	40
6	41	test5@test.com	test5	test5	$2a$10$3PJqV…	<null>
7	49	test4@test.com	test4	test4	$2a$10$ghTSB…	<null>

fig2.5.1.1: user table

Register

Please fill out this to register

First Name

bob

Last Name

bob

Email address

test@test.com

You email will be used for your login.

Password

••••••••

Click here to Register

Already have an account Login here

fig2.5.1.2: trying to register test@test.com

Register

Please fill out this to register

First Name

first name

Last Name

last name

Email address

email

You email will be used for your login.

Password

password

That email address has already been used. Please correct register with a different email address.

Click here to Register

Already have an account Login here

fig2.5.1.3: registration failed

When the customer tries to register test@test.com, the system functions as expected and rejects this register attempt(*the email exists already*)(*fig2.5.1.3*). The test succeed with the correct result.

2.5.3. Login Component

This section describes one test case for the the login component of the system. The test will try and log in a user with a email and password combination that is not saved in the users table(*fig2.5.3.1*). The expected result of this test is that the system will deny login access.

	user_id	email	first_name	last_name	password	customer_profile_id
1	1	christobelchan@gmail.com	chris	chan	$2a$10$AVWm/…	<null>
2	2	test@test.com	test	test	$2a$10$Cp05h…	1
3	17	test1@test.com	test1	test1	$2a$10$2qUPA…	37
4	18	test2@test.com	test2	test2	$2a$10$7OTik…	38
5	39	test3@test.com	test3	test3	$2a$10$RTjld…	40
6	41	test5@test.com	test5	test5	$2a$10$3PJqV…	<null>
7	49	test4@test.com	test4	test4	$2a$10$ghTSB…	<null>

fig2.5.3.1: user table

The email bob@bob.com is entered into the form with a password of 87654321(*fig2.5.3.2*). The result of this action should be that the login attempt is rejected.

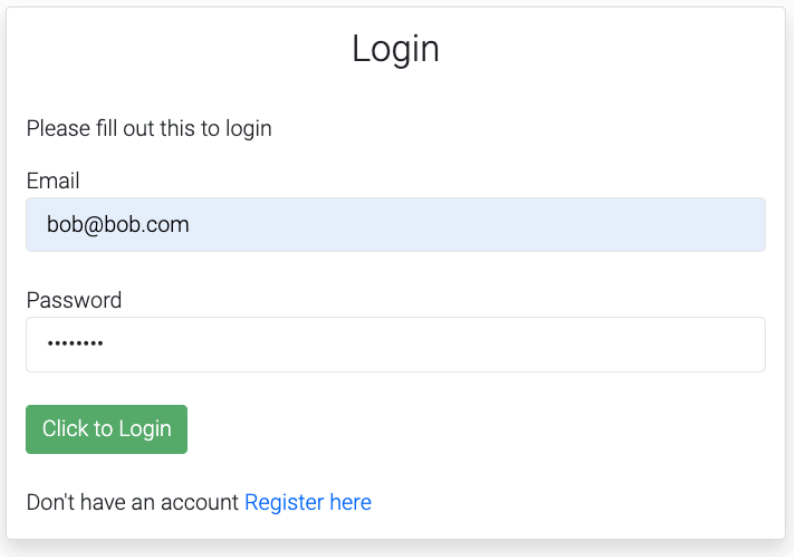

fig2.5.3.2: trying to login with bob@bob.com and password 87654321

In the image(*fig2.5.3.3*) we can see the system functions as expected as a user with email of bob@bob.com and password of 87654321 doesn't exist on the system. The test succeed with the correct result.

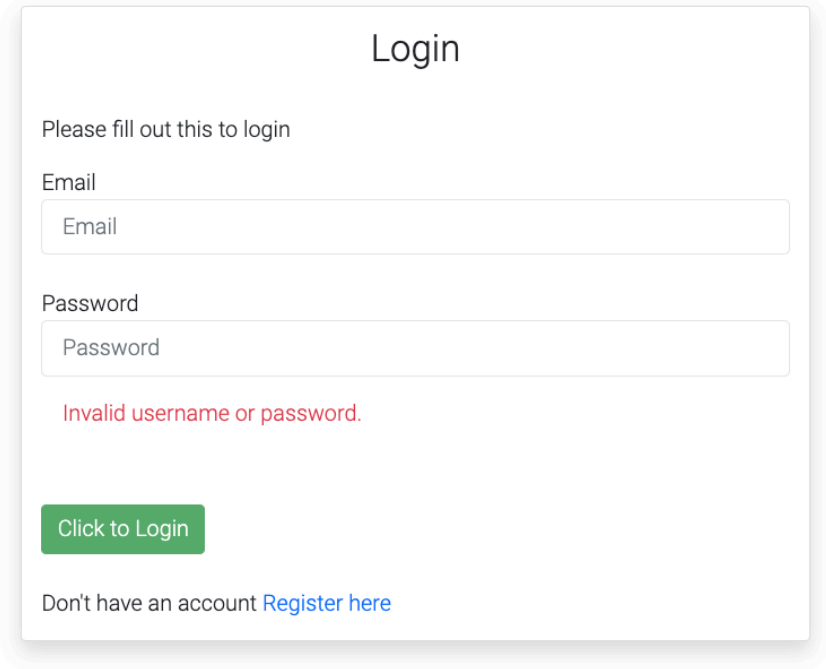

fig2.5.3.3: login failed

2.5.4. Add to Cart Component

This section describes one code test case that was carried for the add to cart function. This test case is to save an item to the cart_items table in the database. The image shows the cart_items table is currently empty.

id	quantity	customer_profile_id	product_id

fig2.5.4.1: cart_items table

The test method(*fig2.5.4.2*) called testSaveItem() declares an object of a CustomerProfile who's id = 1 and a Product who's id = 3. By creating a new CartItems instance, adding the CustomerProfle, Product and a Quantity, we can test if the item gets added to the cart_items table. The method expects that if the CartItem has a cart_item_id(*automatically generated when added to table*), that the item has been added successfully to the table. This would yield a successful test result.

```
@Test
public void testSaveItem(){
    //set the customer Profile and the Product Id
    Integer customerProfileId = 1;
    Integer productId = 3;

    //use the entityManager to get the customer Profile
    CustomerProfile customerProfile = entityManager.find(CustomerProfile.class, customerProfileId);
    //use the entityManager to get the product
    Product product = entityManager.find(Product.class, productId);

    //make new item and set the attributes
    CartItem newItem = new CartItem();
    newItem.setCustomerProfile(customerProfile);
    newItem.setProduct(product);
    newItem.setQuantity(1);

    //save the item
    CartItem savedItem = cartItemRepository.save(newItem);
    //assert that it exists
    assertThat(savedItem.getId()).isGreaterThan(0);
}
```

fig2.5.4.2: CartTest.java testSaveItem()

When running the method we can see it has completed successfully(*fig2.5.4.3*). The entry to the cart_items table also reflects this(*fig2.5.4.4*). The expected result was achieved.

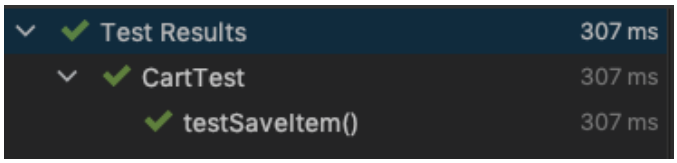

fig2.5.4.3: successfully test

	id	quantity	customer_profile_id	product_id
1	51	1	1	3

fig2.5.4.4: the cart_items table updated with new entry

2.5.5. Upload Prescription Component

This section describes a test case that was used when testing the upload prescription functionality of the application. In this test case(*fig2.5.4.5*) a registered customer, test6, inputs their customer profile information, including uploading their eye prescription. They have chosen an image with the name prescriptionTest6.jpeg. When all the fields are entered and an image is selected, the system is expected to add the customer profile and therefore upload the prescription without issue.

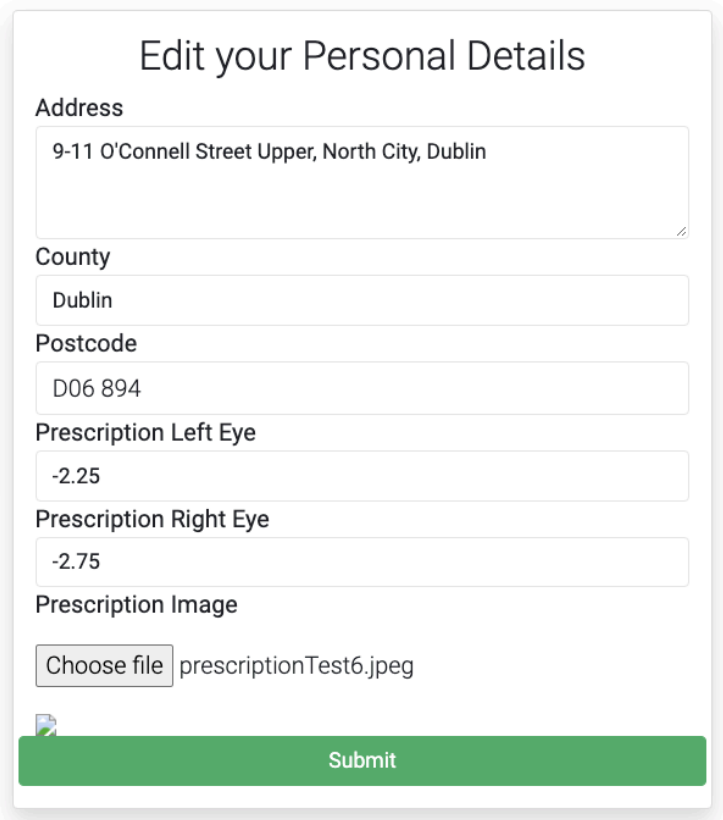

fig2.5.5.1: inputting all the fields of the customer profile

To run the test the submit button was pushed. The system redirects to the profile page where we can see the full customer profile and all the details have been successfully submitted.

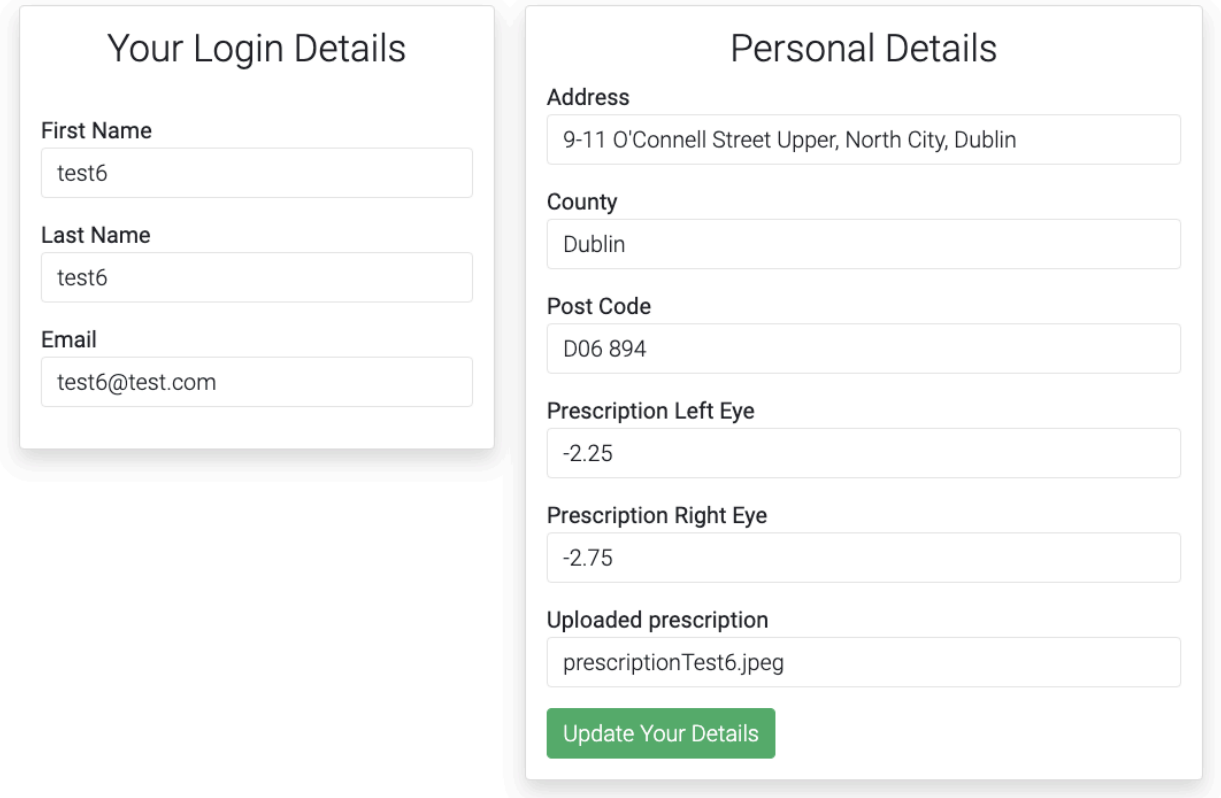

fig2.5.5.2: The customer profile including prescription have been uploaded.

If we check the system we can see that a new entry has been added to the customer_profile table(*fig2.5.5.3*). Also the file prescriptionTest6.jpeg has been saved to the correct directory on the system(*fig2.5.5.4*). The test ran as expected.

	id	address	...	email	...	postcode	prescription_im...	right_eye
1	1	190 The New Hardwicke …	Dublin	test@test.com	-2.25	D07 XY4	test prescription.jpeg	-4.25
2	37	Mayor Street Lower, In…	Dublin	test1@test.com	-1.75	D07 XY4	prescriptiontest2.jpeg	-2
3	38	Dublin City University…	Dublin	test2@test.com	-2	D07 XY4	prescriptionTest3.jpeg	-1.75
4	40	62 O'Connell Street Up…	Dublin	test3@test.com	-1.25	D01 TV2	prescriptionTest4.jpeg	-1.5
5	53	9-11 O'Connell Street …	Dublin	test6@test.com	-2.25	D06 894	prescriptionTest6.jpeg	-2.75

fig2.5.5.3: The customer_profile table

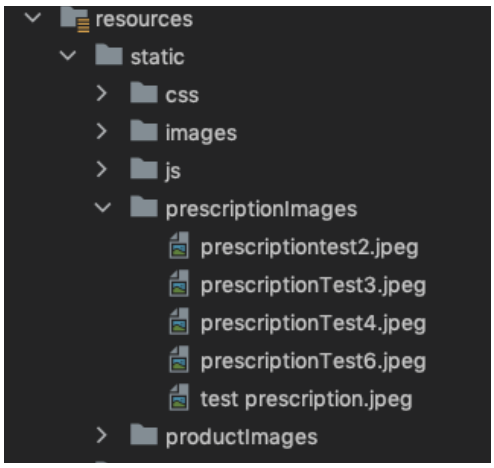

fig2.5.5.4: file successfully uploaded

2.5.6. Add Payment Component

This section describes one test case for the add payment component of the application. This test inputs random fields for the card details. The card details that will be entered are:

Card number: 1234 5678 9876 5432

Card expiry date: 02 / 12

Card CVC: 3232

The image(*fig2.5.6.1*) shows the details listed above entered to the payment.html form. The system is expected to reject this card number from being processed.

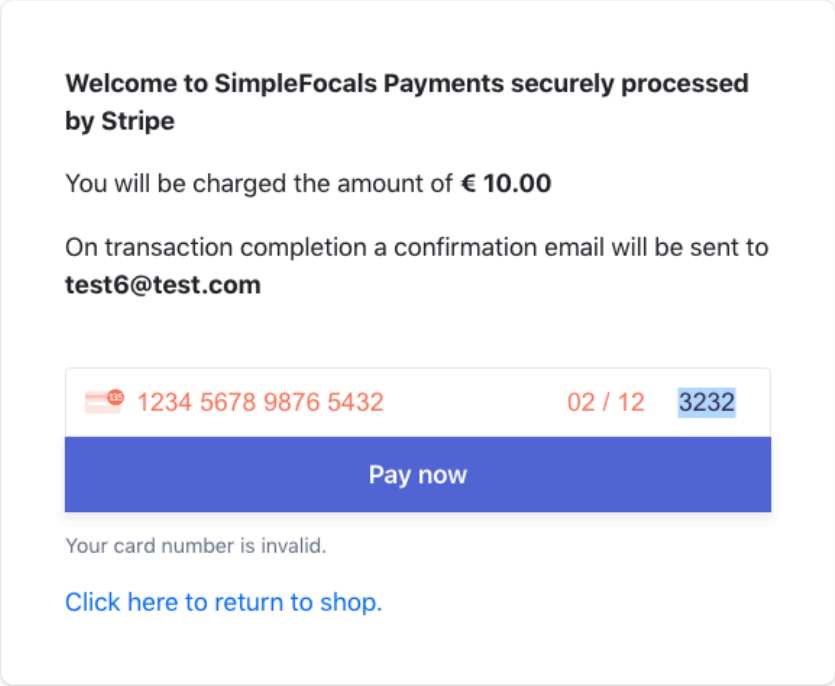

fig2.5.6.1: random card number inputted to payment

Clicking on the Pay now button doesn't allow payment. The form shows that the card number is invalid. The test ran as expected, the input does not yield a payment through Stripe.

2.5.7. Add product Component

This section describes one test case for the add product functionality of the application. This will be a manual test where an administrator inputs fields of a product but leaves one field empty. The administrator will input the fields of name, category, price and choose a product Image. They will leave the product description blank(*fig2.5.7.1*). After pushing submit, it is expected not allow the product to be added as all fields must be inputted

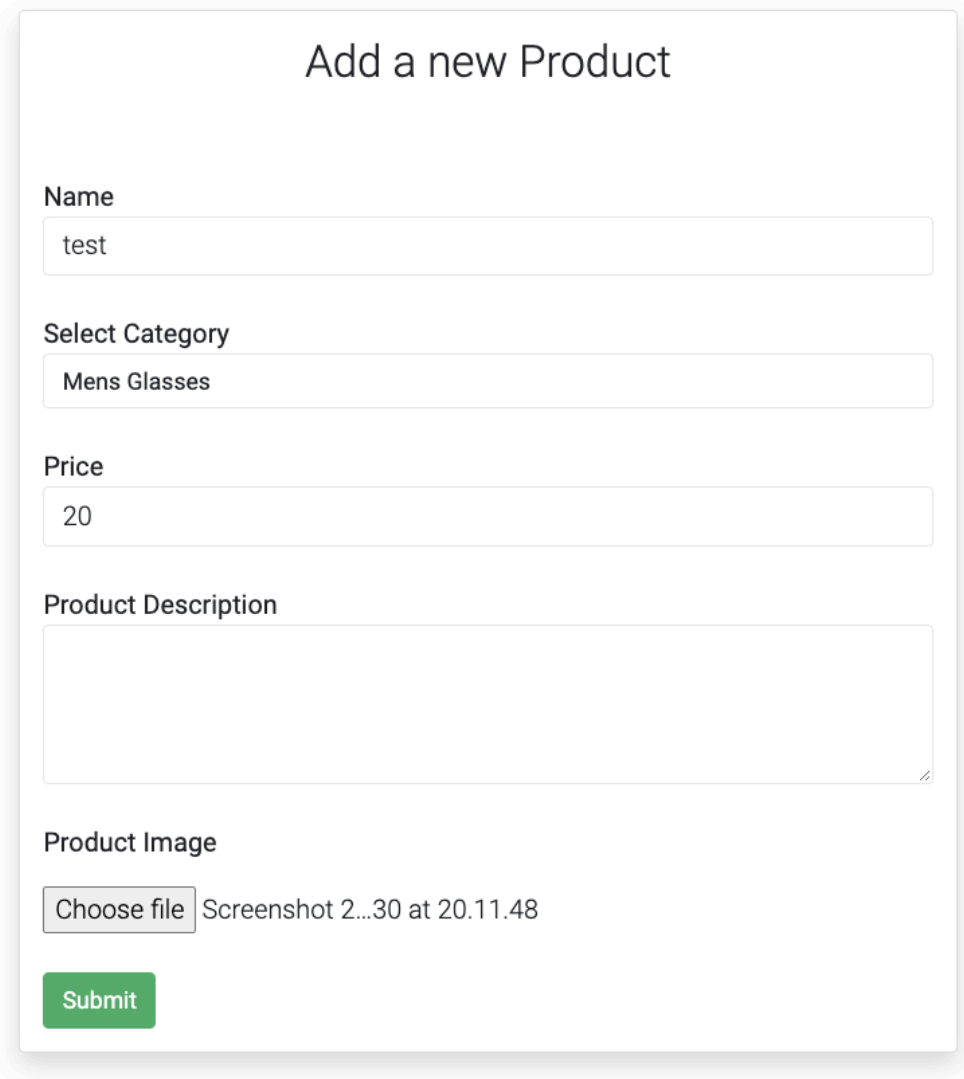

fig2.5.7.1: Add product with the product description field blank

Pushing the submit button yields the warning that the product description must be filled in(*fig2.5.7.2*). This is the expected result of the test. The test ran successfully.

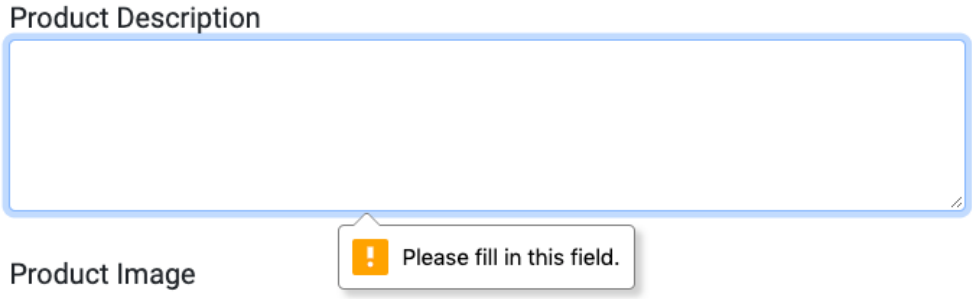

fig2.5.7.2: the product description must be added.

2.6. Graphical User Interface (GUI) Layout

This section describes the graphic user interface layout of the application. The plan for the GUI layout was first outlined in the ***Product Design Specification Document - 3.1 Logical View.*** The GUI was designed with a mobile-first approach to design. This approach was taken because developing the GUI with limited screen real estate distills the elements on the webpage down to only what is most important. It also keeps the web application's GUI consistent across multiple devices and form factors. The image(*fig2.6.1*) shows the site-view of the web application for a customer's view.

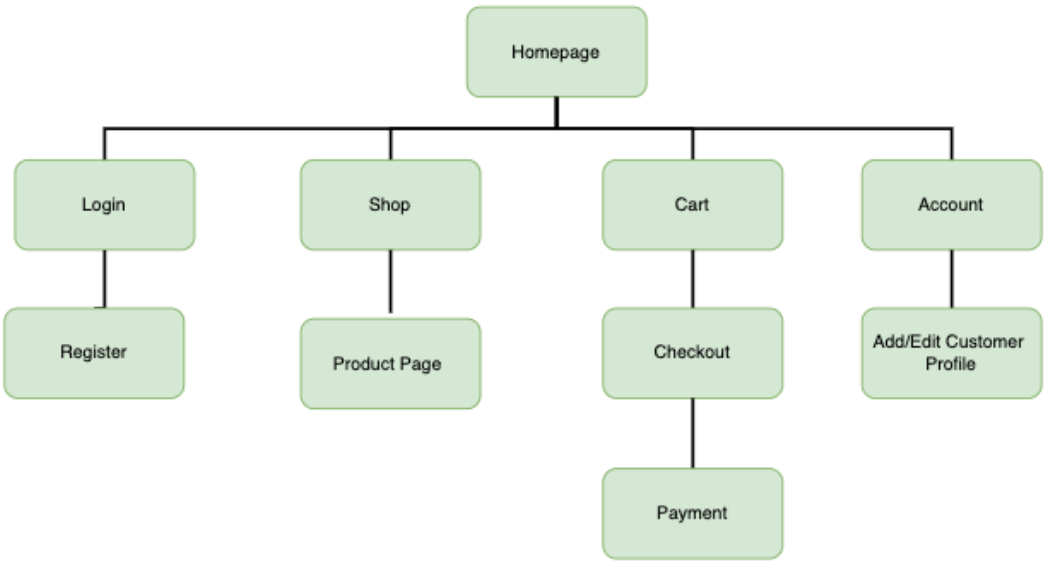

fig2.6.1: Simple site-view for the customer of SimpleFocals

The image(*fig2.6.2*) shows the site-view for administrators view of the web application.

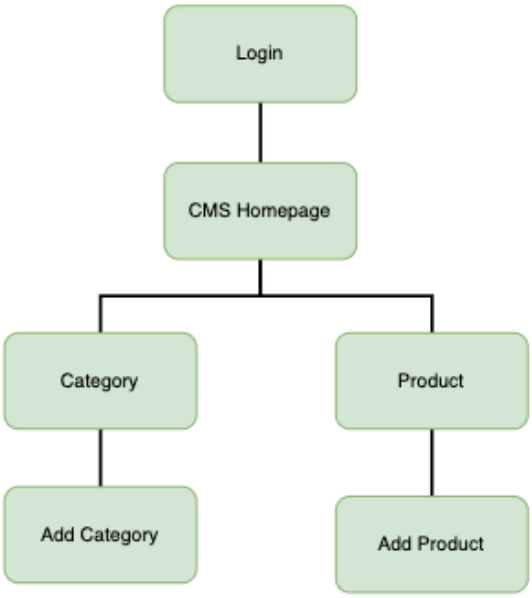

fig2.6.2: Simple site-view for the administrator of SimpleFocals

The GUI was developed using bootstrap classes in combination with HTML5, CSS Javascript, with some design and colour scheme inspiration from templatemo.com. The icons that have been used throughout the website are from fontawesome.com. The website is designed where the majority of body content is displayed on cards with light shadowing. This was a conscious design decision as brings the content the forefront instead of looking flat on the page. All pages of the website have a navigation bar at the top and a footer at the bottom for easy access to important links. The mobile view for the navigation bar for non-registered customer(*fig2.6.3*) and logged in customer(*access to cart and profile*)(*fig2.6.4*) can be seen. The image shows the mobile view for the footer(*fig2.6.5*). The desktop view for the navigation bar for non-registered customer(*fig2.6.6*) and logged in customer(*access to cart and profile*)(*fig2.6.7*). The image(*fig2.6.8*).shows the desktop view of the footer.

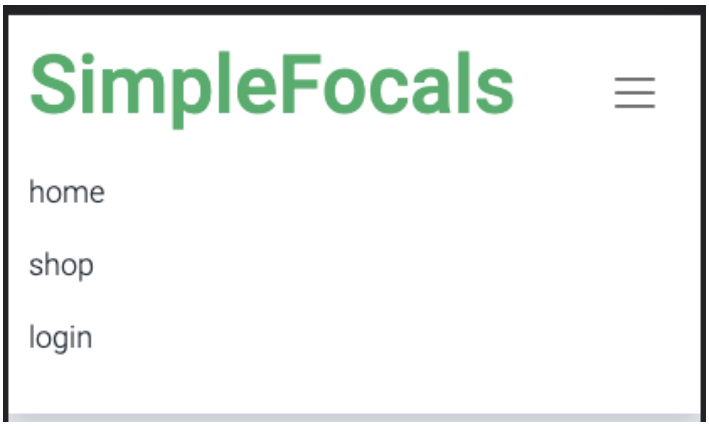

fig2.6.3: nav-bar - mobile: non-registered customer

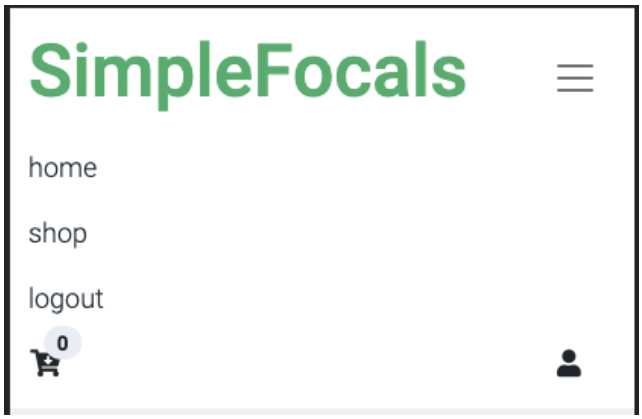

fig2.6.4: nav-bar -mobile: logged in customer

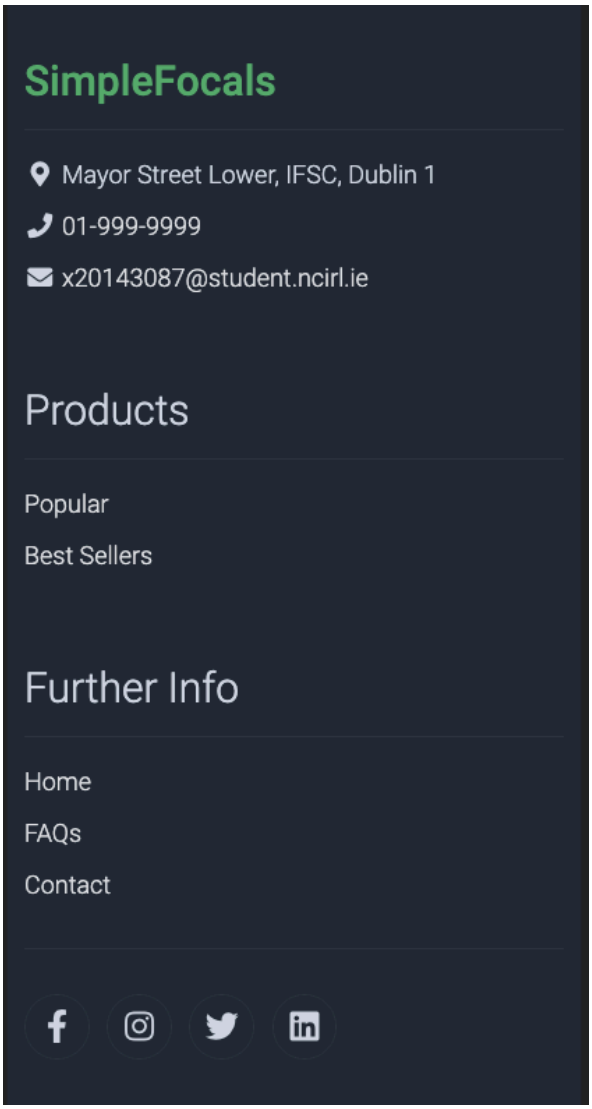

fig2.6.5: footer - mobile

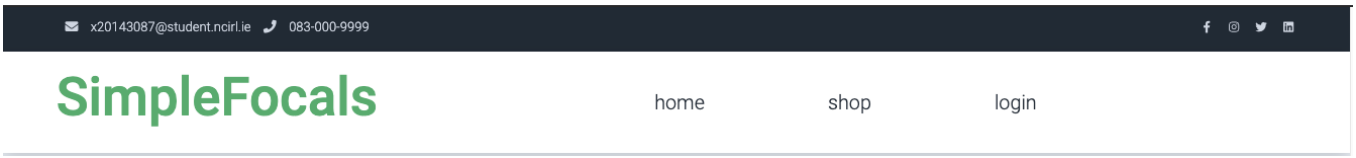

fig2.6.6: Nav-bar - Desktop: non-registered customer

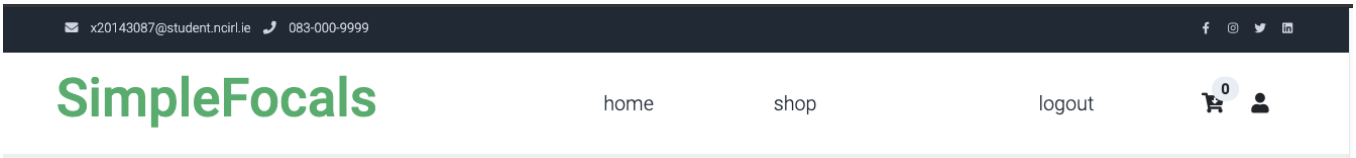

fig2.6.7: Nav-bar - Desktop: logged in customer

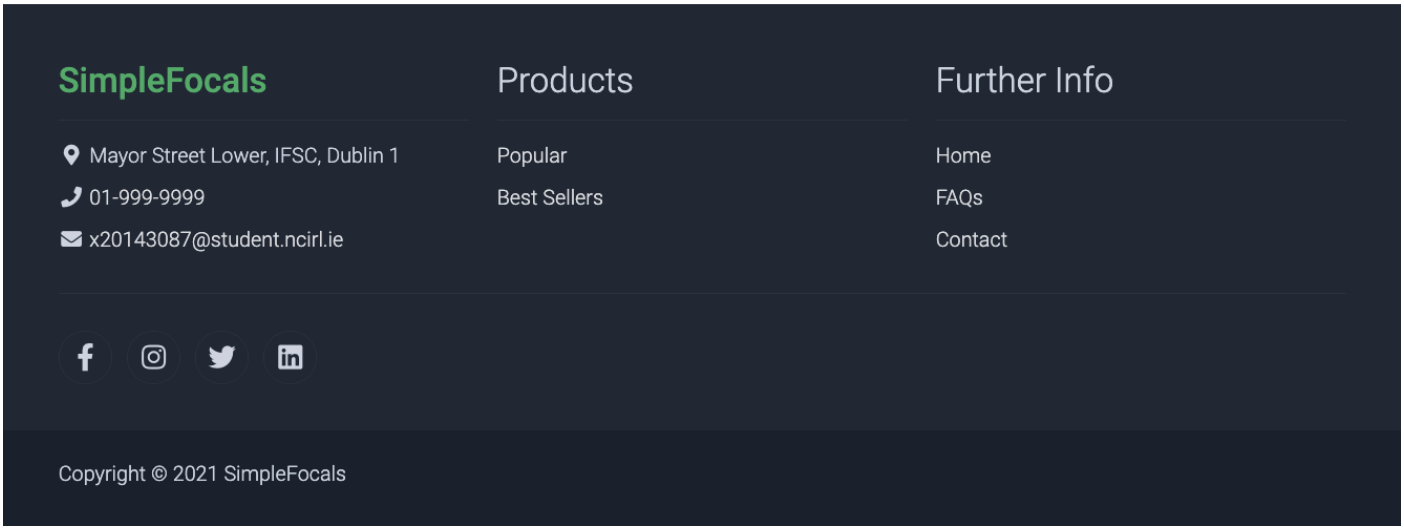

fig2.6.8: Footer - Desktop

2.6.1. Homepage

The homepage has three sections. The carousel front banner, the simple steps section and the featured eyewear section. The carousel rotates three images and tells of SimpleFocals value proposition. I believe the carousel for the front-page of the web application is a good design choice because it dynamically rotates different content in a fixed area saving space on the page. The simple steps section describes the steps involved in ordering from SimpleFocals. This section was consciously put here in order to inform the customer of the steps to follow when ordering. The hover-over transition on each block is a nice touch in keeping with the web applications colour design language. The featured eyewear section shows the most popular eyewear available at the store. This section is here to strike the customer's curiosity and give a preview of the kinds of items they can purchase in the store. The mobile view of the homepage is shown in the images(*fig2.6.1.1*)(*fig2.6.1.2*)(*fig2.6.1.3*)(*fig2.6.1.4*). The desktop view of the web application is shown in the images(*fig2.6.1.5*)(*fig2.6.1.6*)(*fig2.6.1.7*).

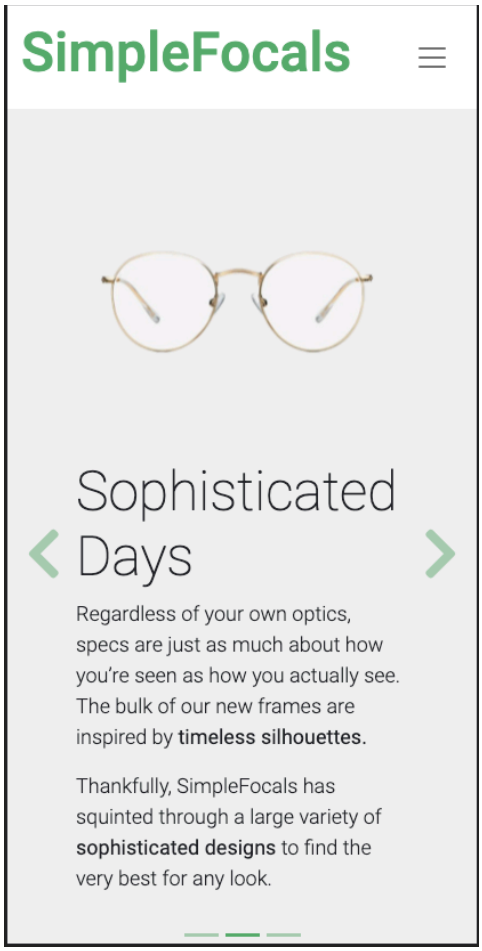

fig2.6.1.1: homepage - mobile

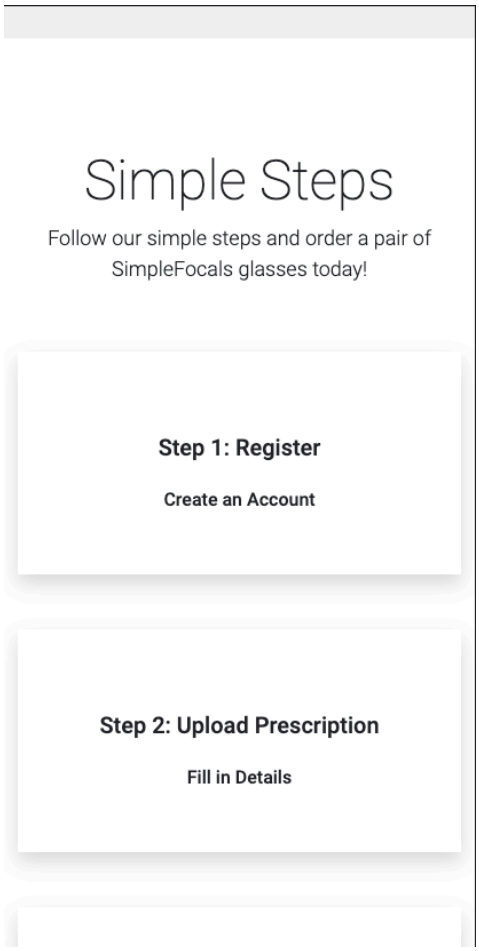

fig2.6.1.2: homepage - mobile

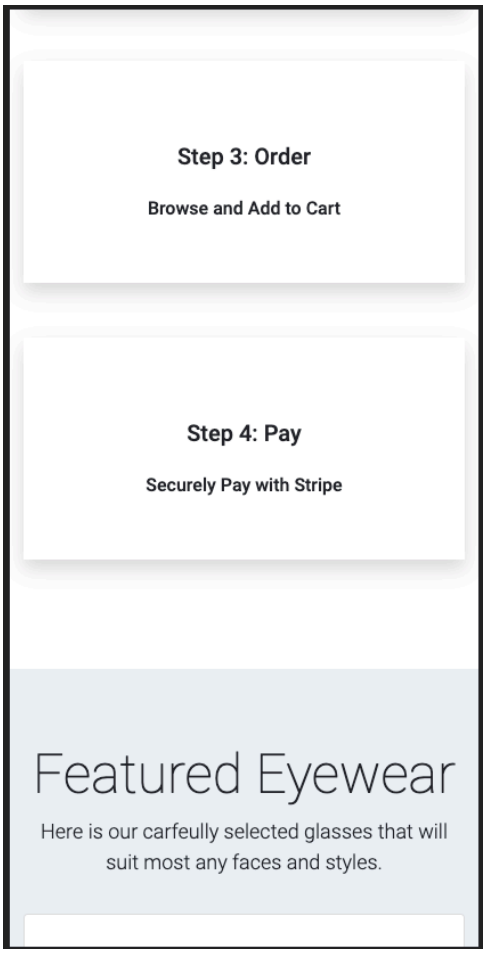

fig2.6.1.3: Homepage - Mobile

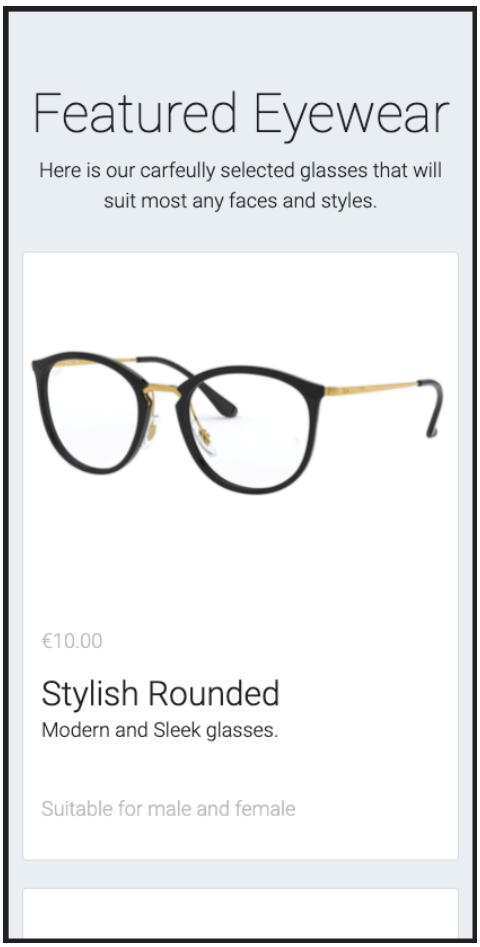

fig2.6.1.4: Homepage - Mobile

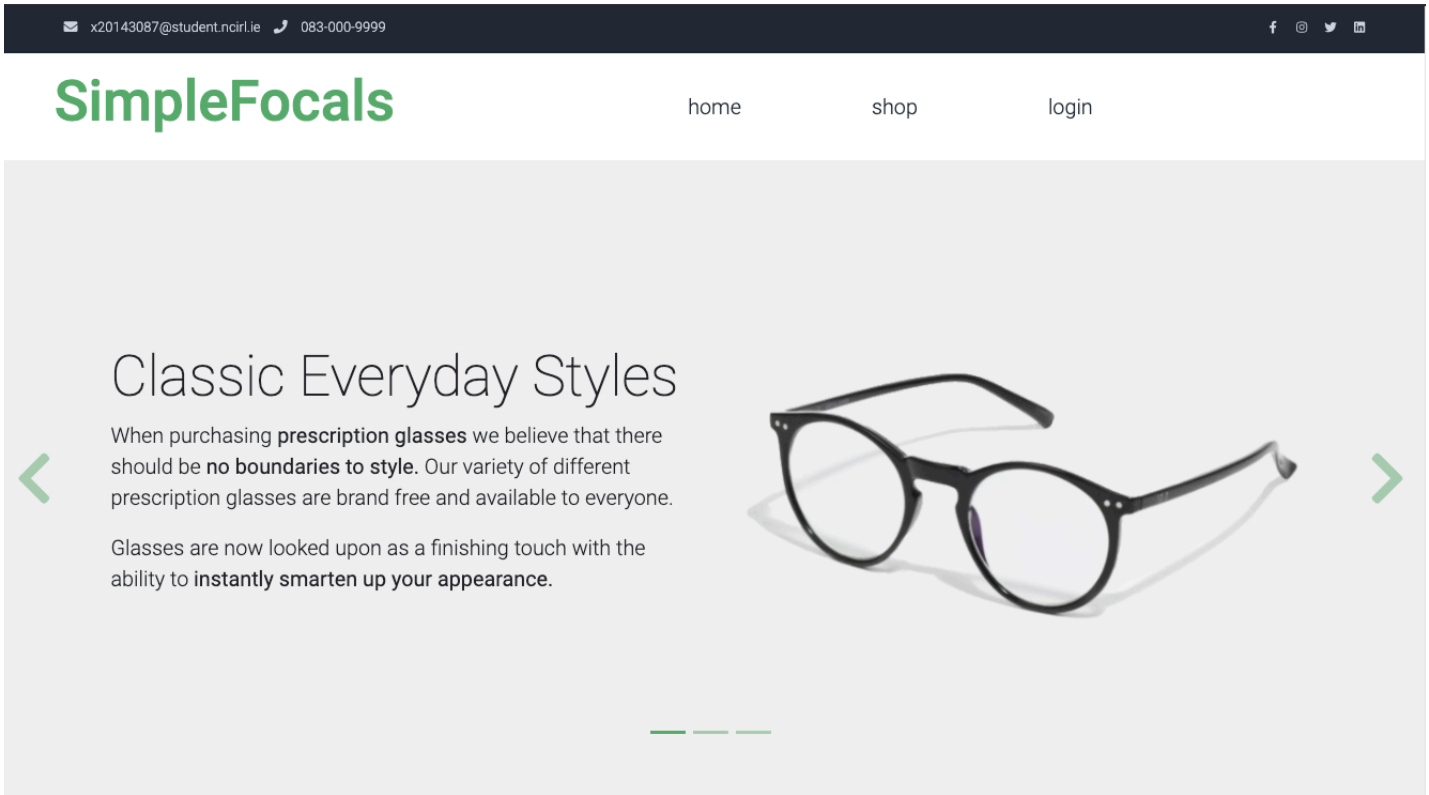

fig2.6.1.5: Homepage - Desktop

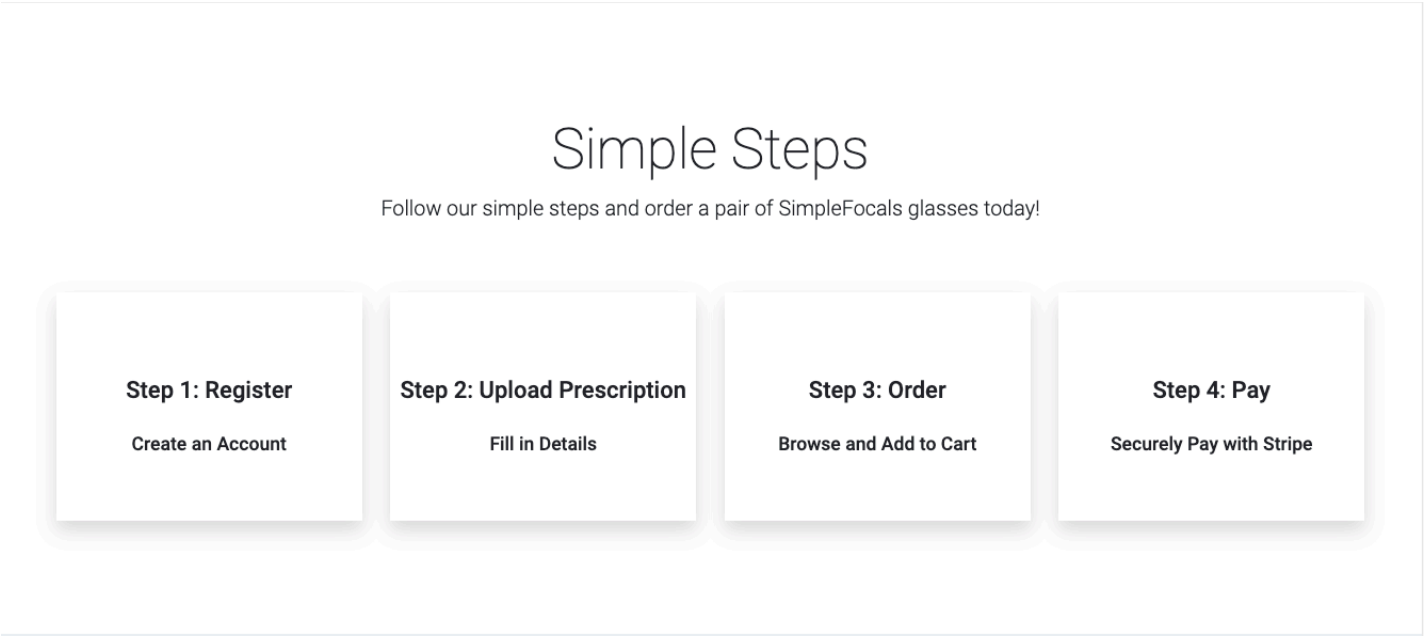

fig2.6.1.6: Homepage - Desktop

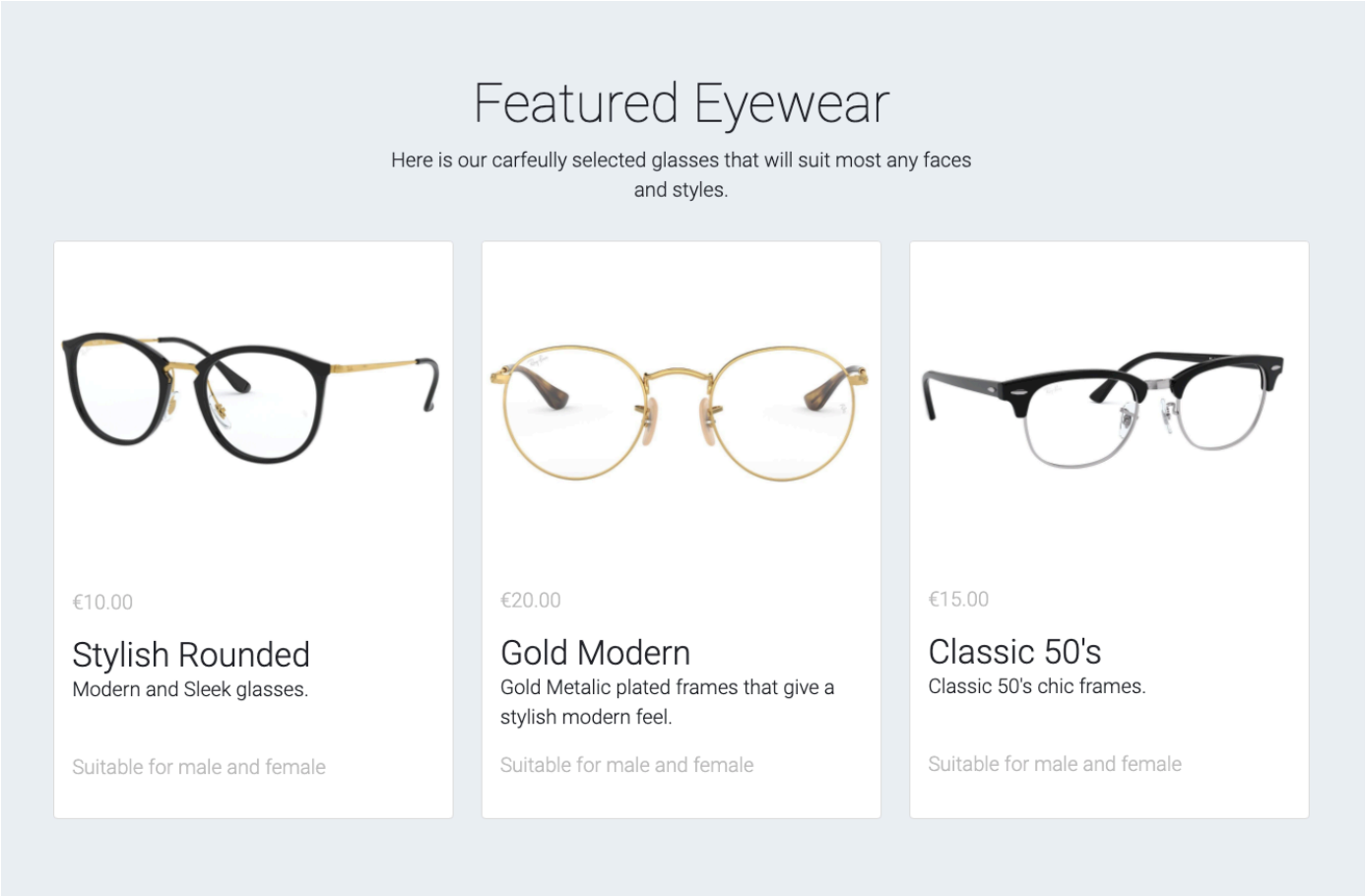

fig2.6.1.7: Homepage - Desktop

2.6.2. Login

The login page has a form displayed on a lightly shadowed card elevating the content. The images shows the mobile view(*fig2.6.2.1*) and the desktop view(*fig2.6.2.1*) of login.

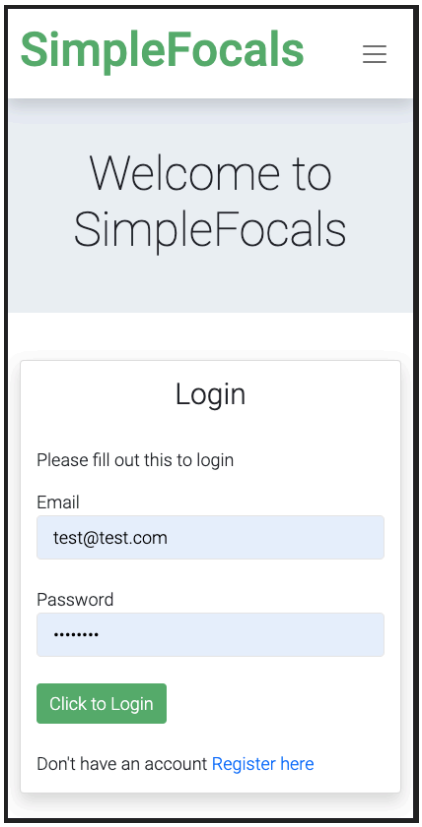

fig2.6.2.1: Login - Mobile

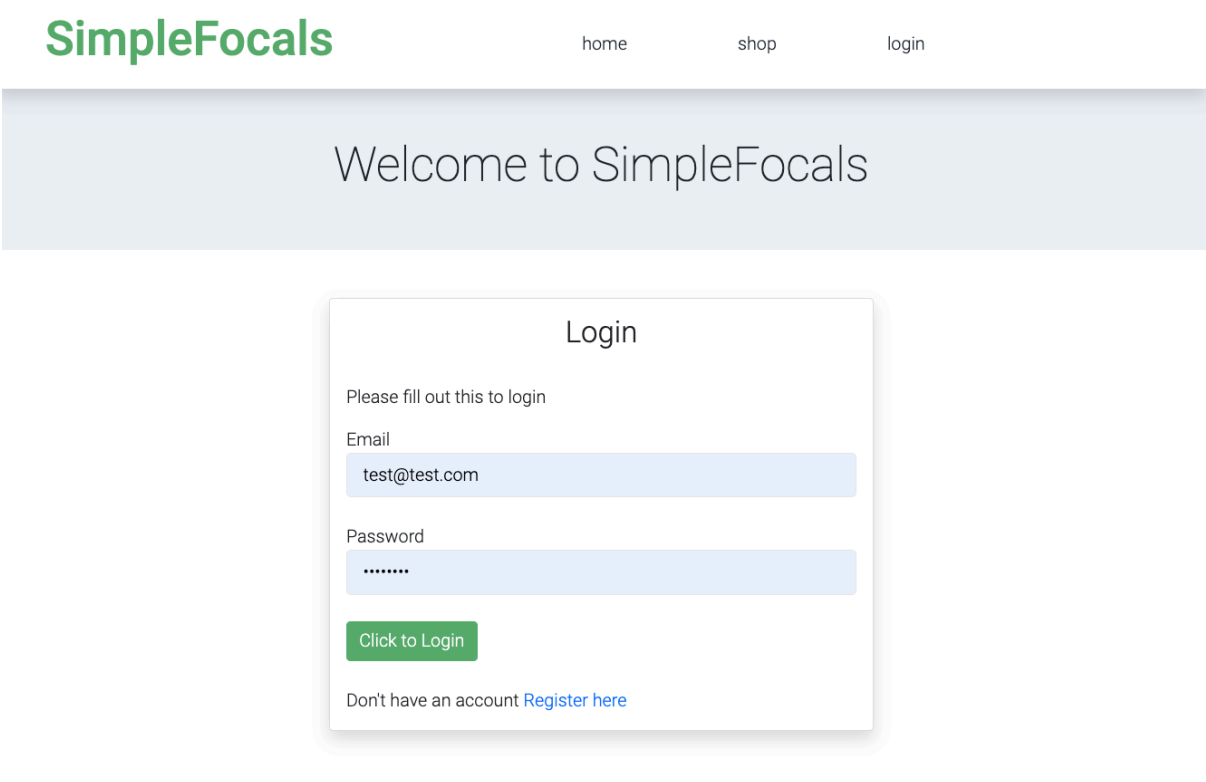

fig2.6.2.2: Login - Desktop

2.6.3. Register

The register page has a form displayed on a lightly shadowed card elevating the content. The Images show the mobile view(*fig2.6.3.1*) and the desktop view(*fig2.6.3.1*) of login.

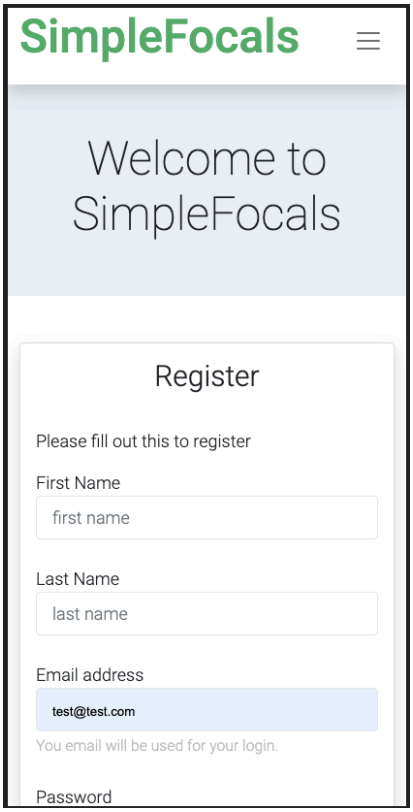

fig2.6.3.1: Register - Mobile

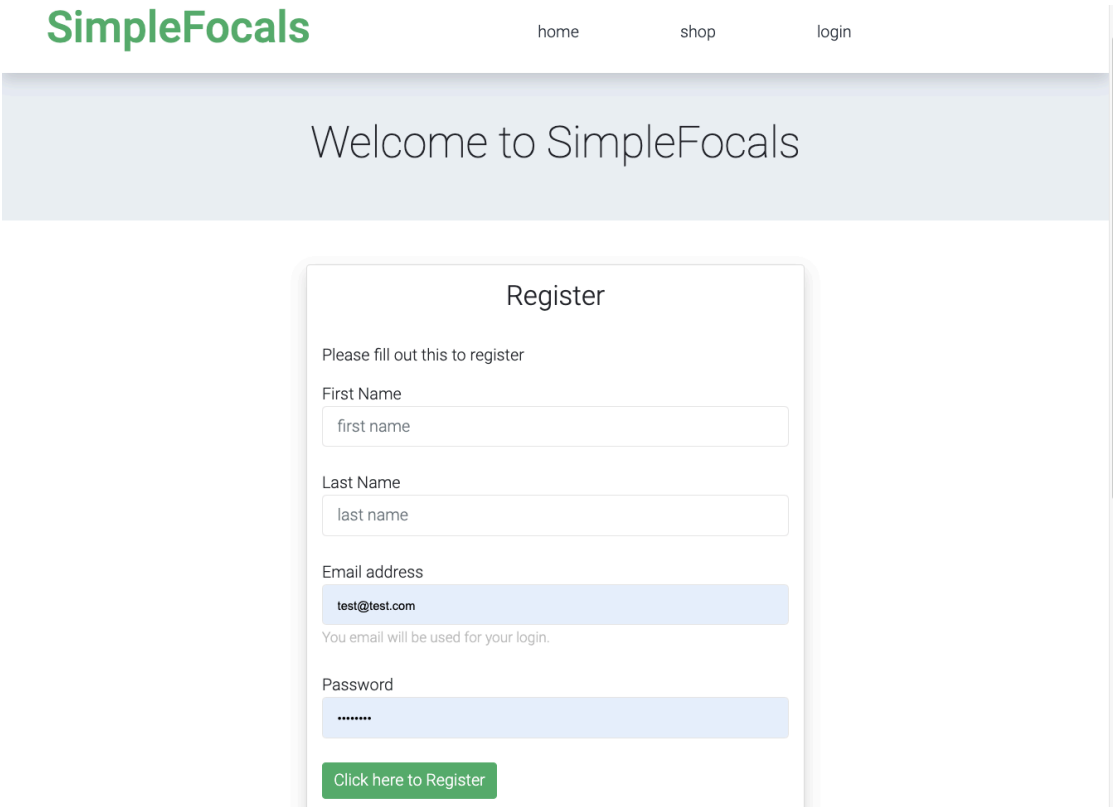

fig2.6.3.2: Register - Desktop

2.6.4. Shop

The shop consists of two pages. The shop and the view product pages. The shop page displays all the glasses available in the store with the ability to filter by category. The filter by category is a nessecary option in any online store. The view product page is the individualised page for a specific product with details about the product and the ability to add to cart. Again all the content in the shop and the individual product pages are displayed on lightly shadowed cards in order to elevate the content bringing it closer to the customer. The image shows the mobile view of the shop page and the view product page. The image shows the desktop view of the shop page and the view product page.

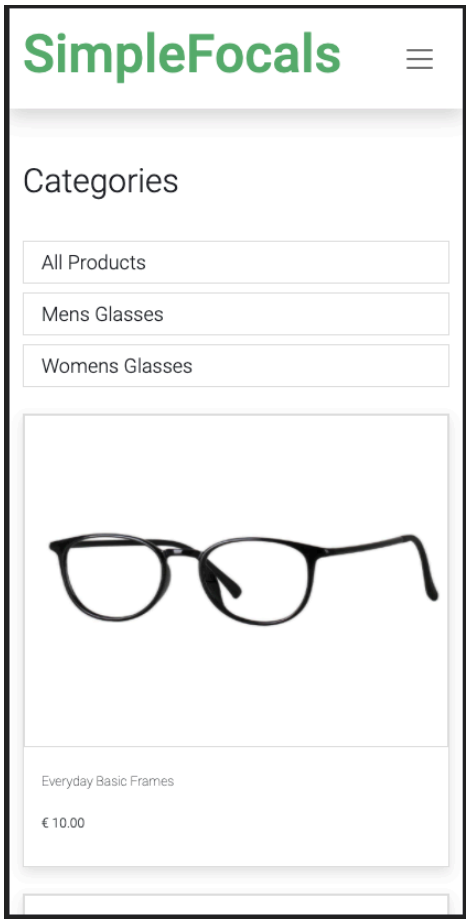

fig2.6.4.1: Shop - Mobile

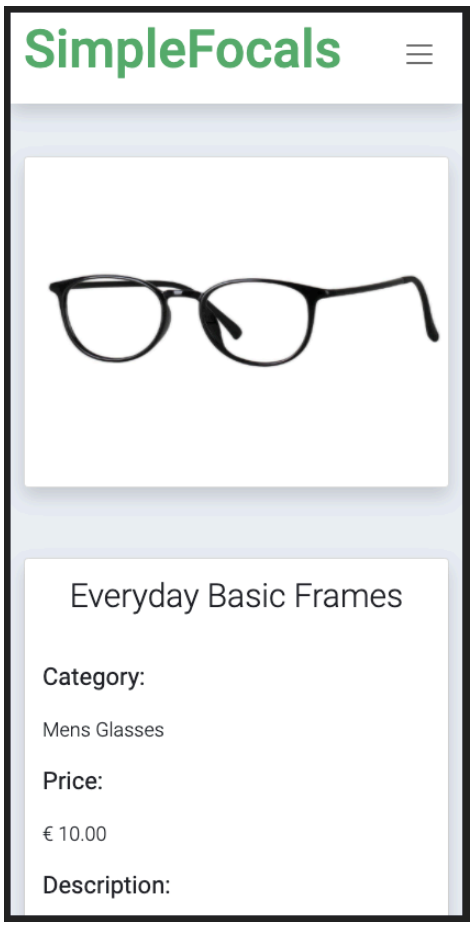

fig2.6.4.2: View product - Mobile

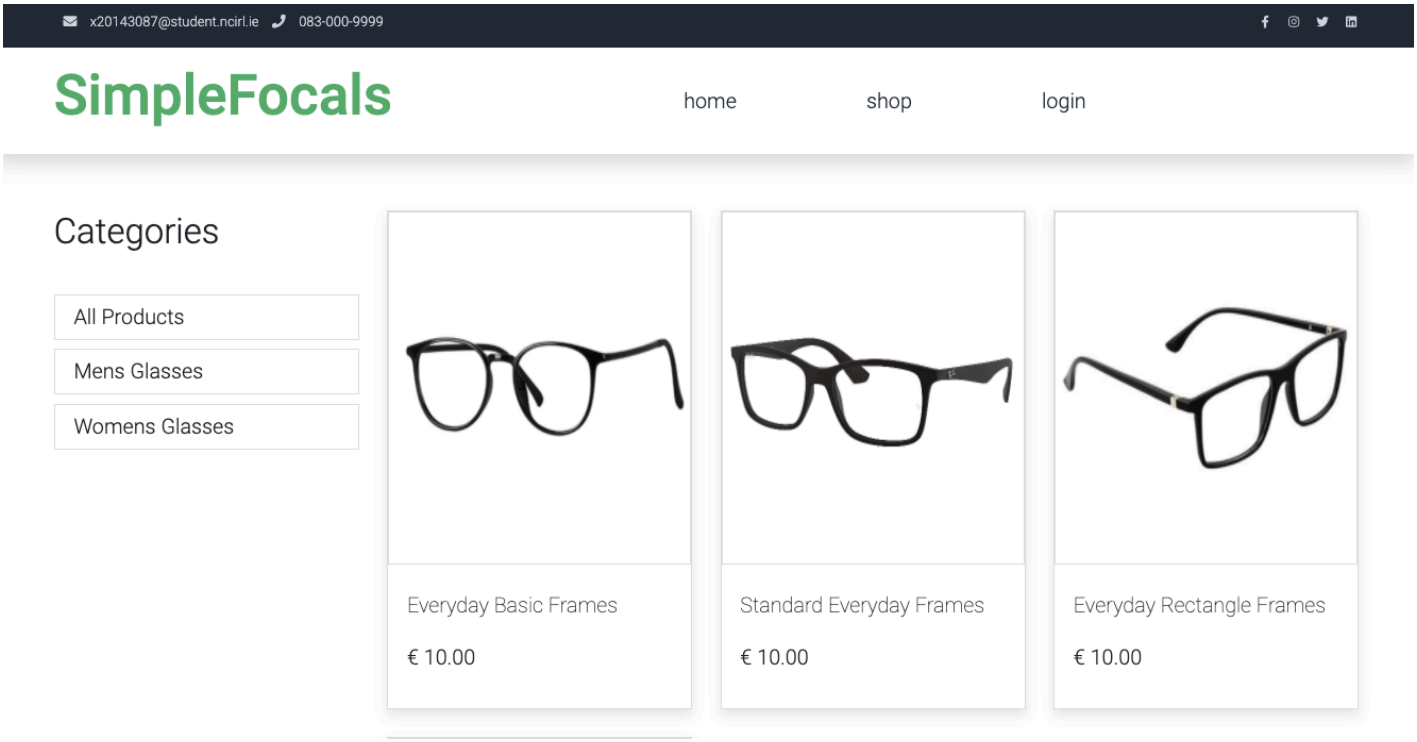

fig2.6.4.3: Shop - Desktop

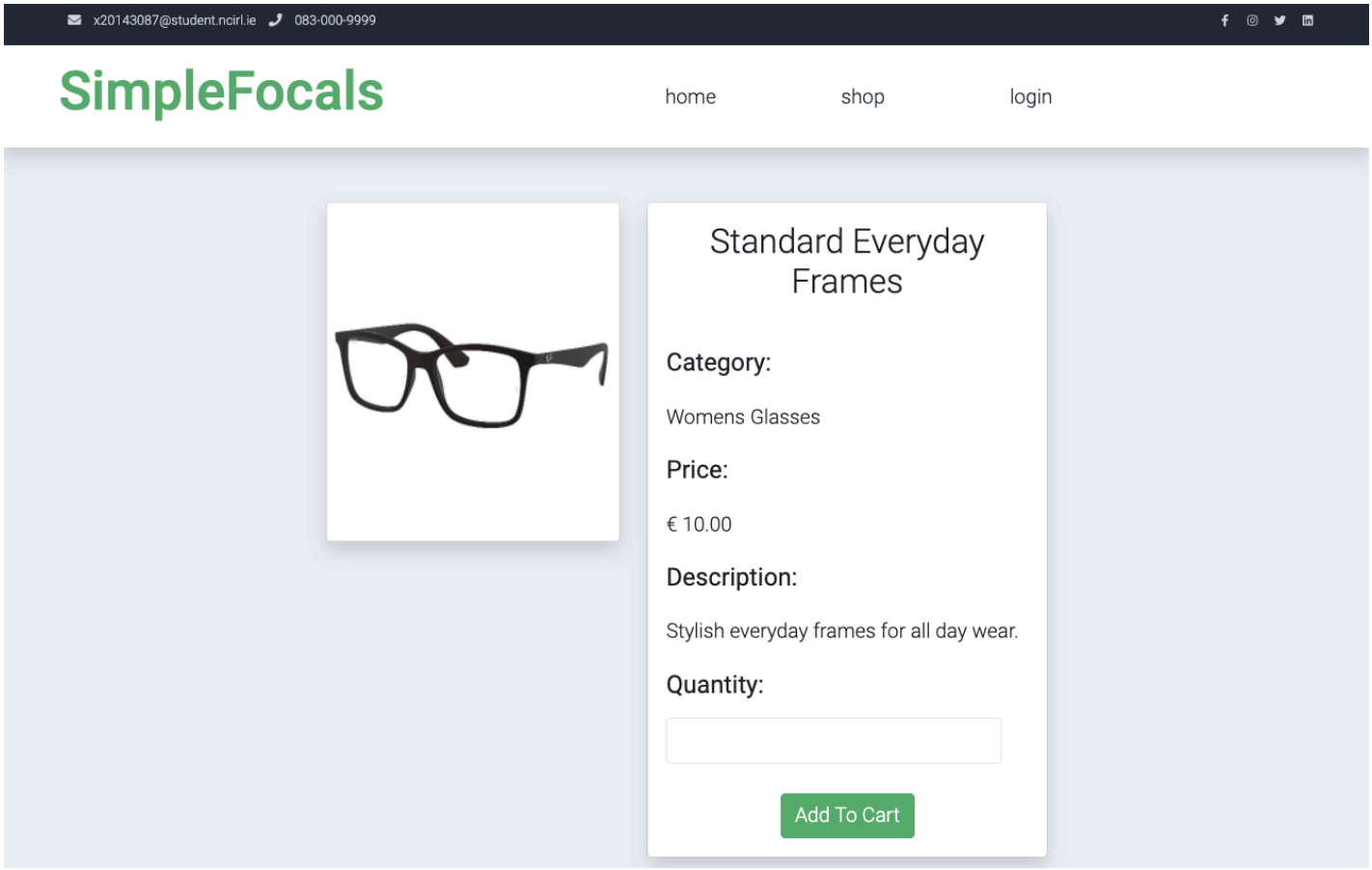

fig2.6.4.4: View Product - Desktop

2.6.5. Cart

The cart page consists of a number of lightly shadowed cards. The items that are added to cart will each have their own individual card. Each card will have a summary of the item and the option to remove the item from the cart. I believe this card implementation of have items in a cart id the most logical way to design a cart. The information is clearly presented and easy to mange. The cart summary of the items and cost is also displayed on its own card with the option to go to checkout. The mobile view of the cart is shown on image(*fig2.6.5.1*)(*fig2.6.5.2*). The desktop view of the cart is shown in the image(*fig2.6.5.3*).

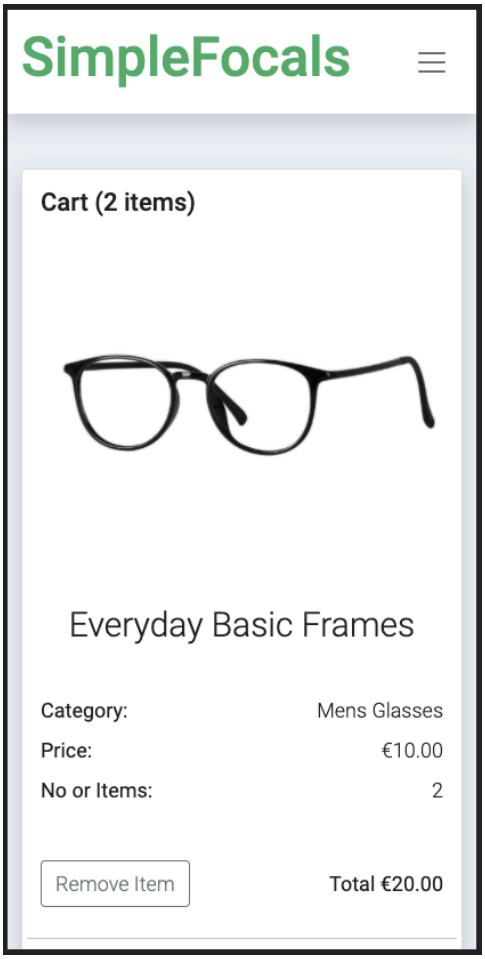

fig2.6.5.1: Cart - Mobile

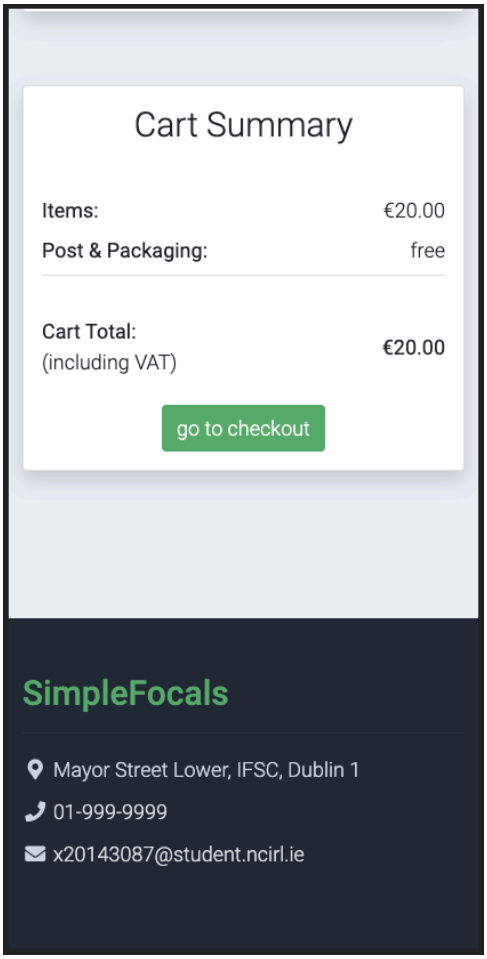

fig2.6.5.2: Cart - Mobile

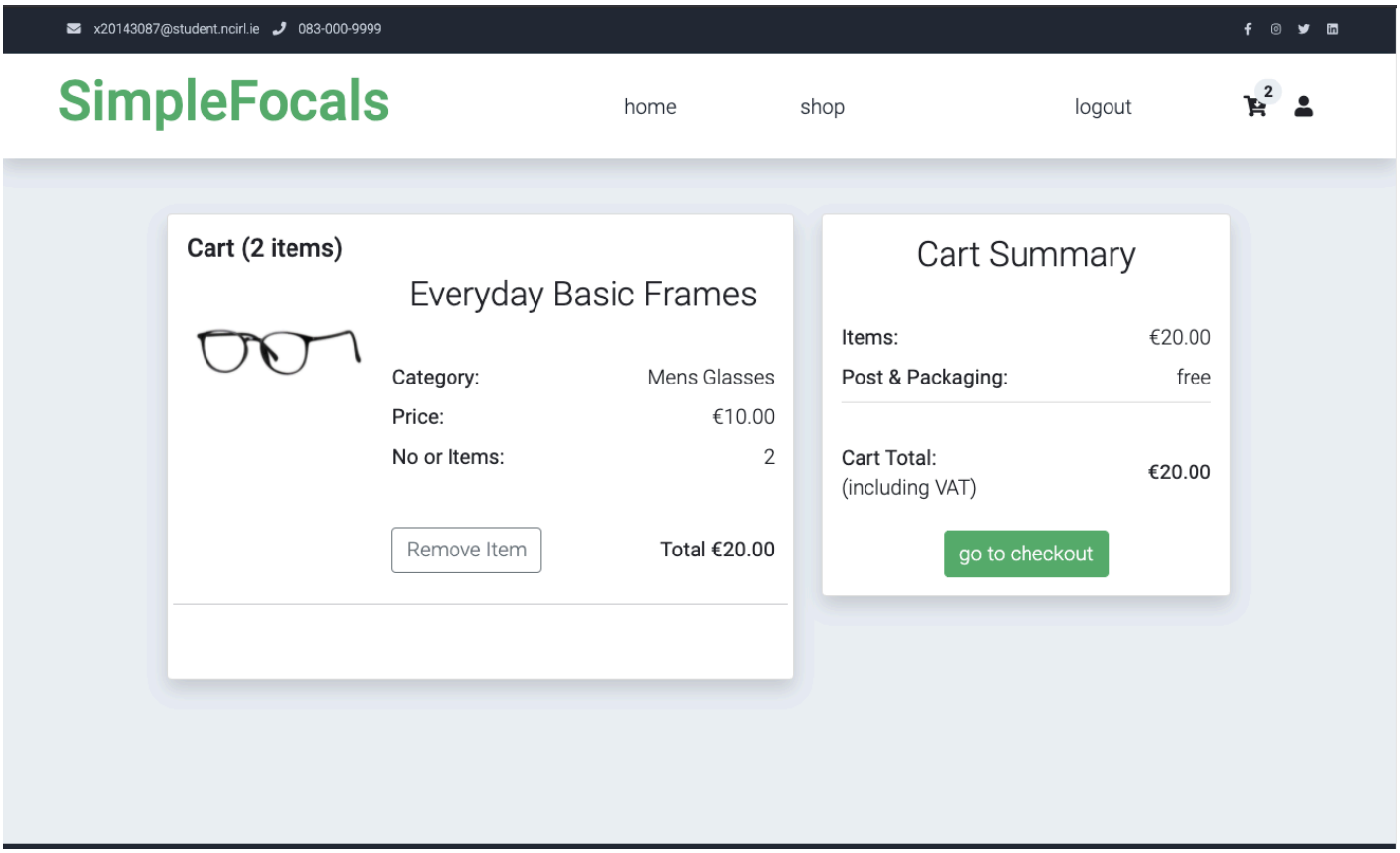

fig2.6.5.3: Cart - Desktop

2.6.6. Checkout

The checkout page consists of two of cards. Again the same theme of shadowed cards is evident here. One card has the customer details including details of the customers prescription while the other card shows the order summary. All the buttons are green, consistent with the whole website colour design language. The images show the mobile view of the checkout page(*fig2.6.6.1*)(*fig2.6.6.2*). The image shows the desktop view of the checkout page(*fig2.6.6.3*).

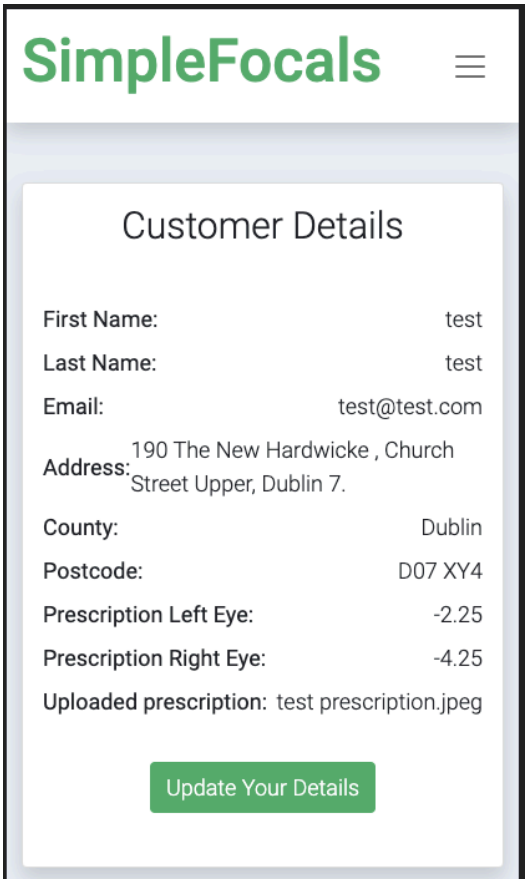

fig2.6.6.1: Checkout - Mobile

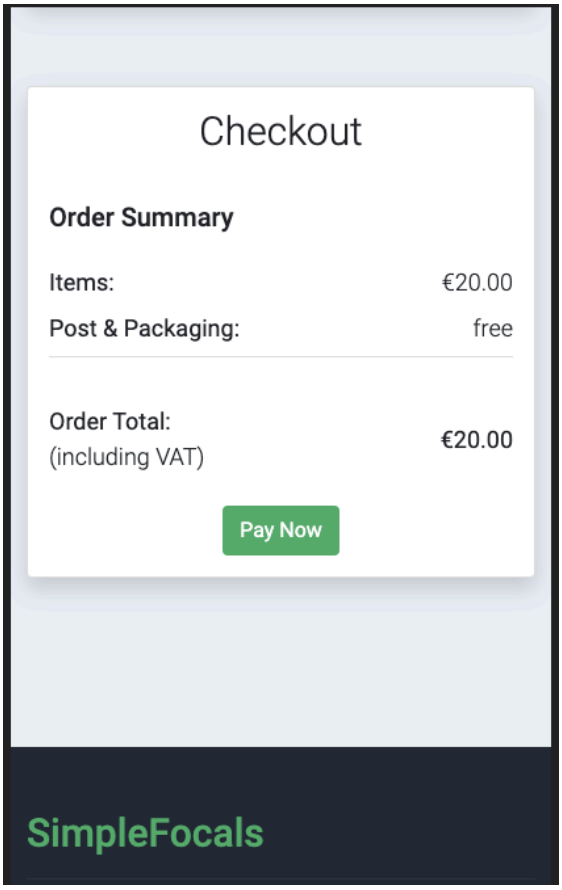

fig2.6.6.2: Checkout - Mobile

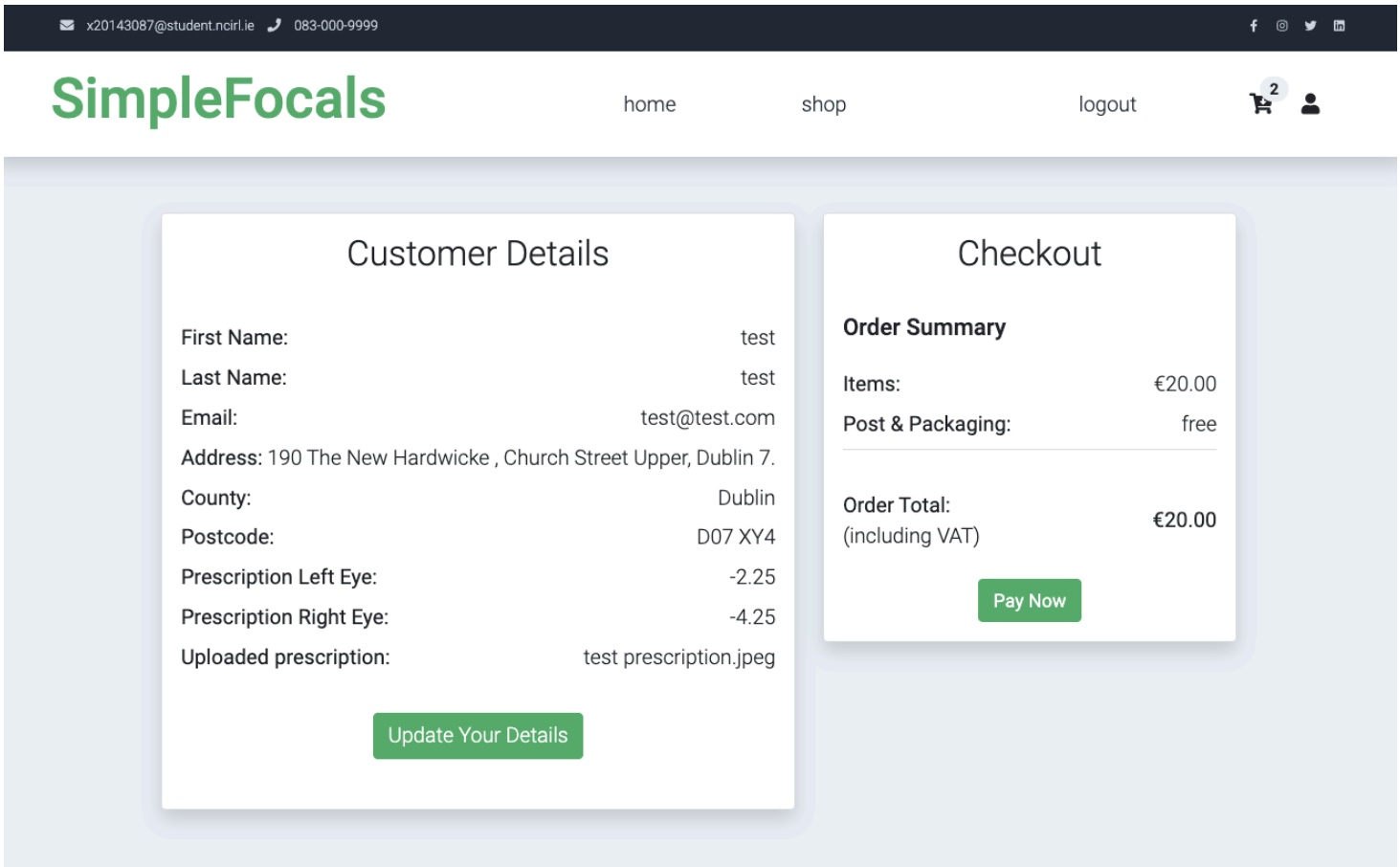

fig2.6.6.3: Checkout - Desktop

2.6.7. Payment

The payment page is where the customer enters their card details to pay fo the items on their order. The page consists of just one form in the centre of the page. Unfortunately at the time of writing only the desktop view of payment page has been optimised. The reason for this is the payment page design and layout is taken directly from Stripe, they have not developed for a mobile view. With the time constraint of the project I have not been able to dedicate time to designing a mobile view. The desktop view of the payment page(*fig2.6.6.3*).

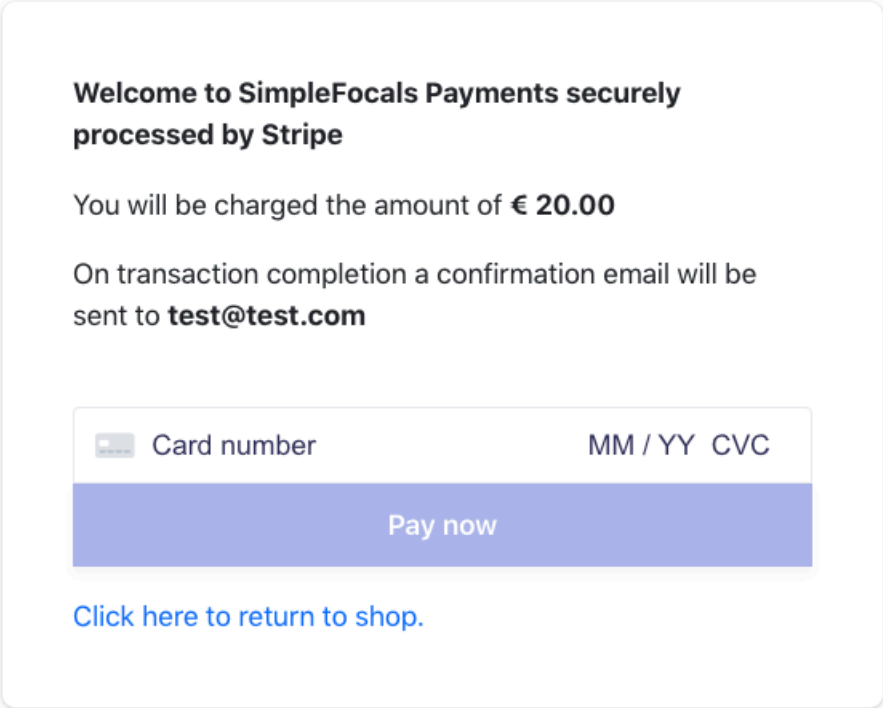

fig2.6.7.1: Payment - Desktop

2.6.8. Profile

The profile section consists of two pages. The profile page and the add/edit profile details. Both the pages have the information about login and customer profile details displayed on the elevated shadowed cards. The images show the mobile view for the profile(*fig2.6.8.1*)(*fig2.6.8.2*) and the add/edit profile details(*fig2.6.8.3*)(*fig2.6.8.4*) pages. The images show the desktop view of the profile(*fig2.6.8.5*) and the add profile details(*fig2.6.8.6*) pages.

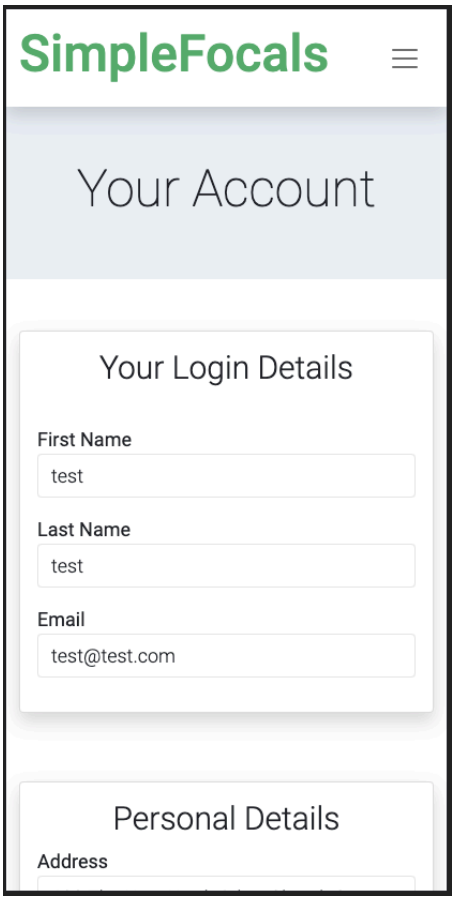

fig2.6.8.1: Profile - Mobile

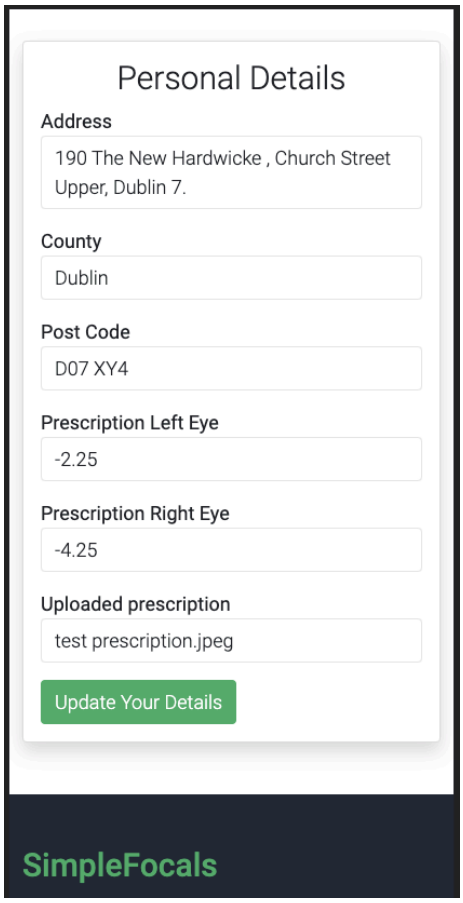

fig2.6.8.2: Profile - Mobile

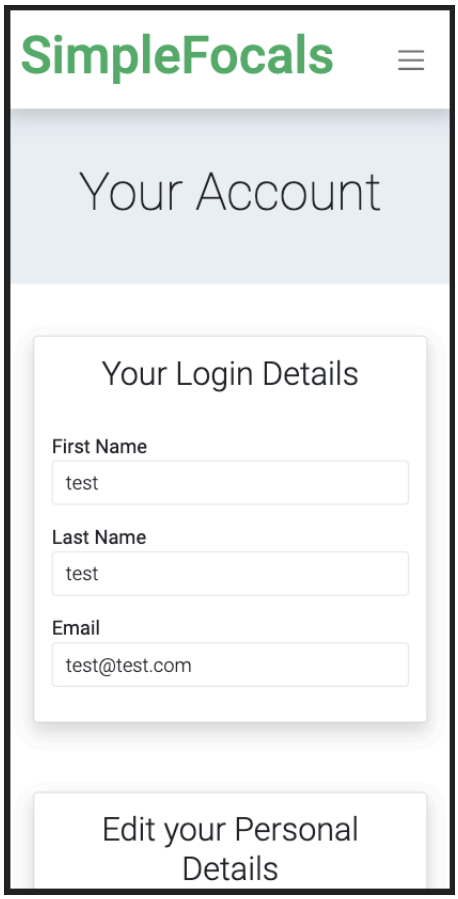

fig2.6.8.3: Add/Edit Profile- Mobile

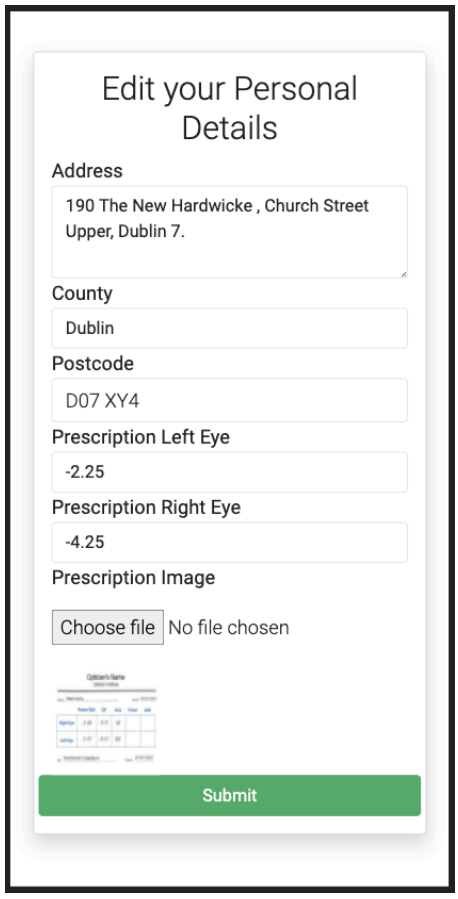

fig2.6.8.4: Add/Edit Profile - Mobile

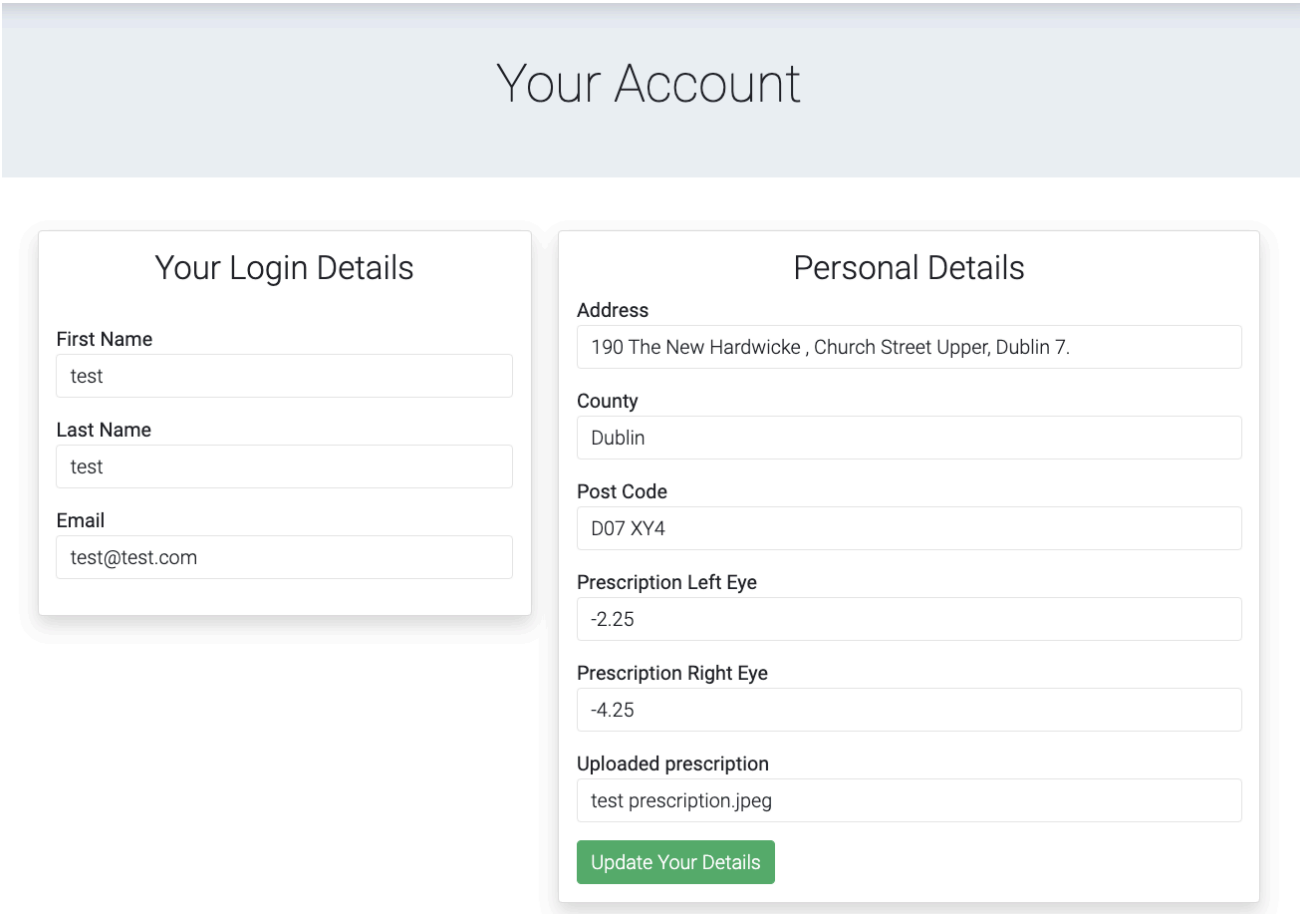

fig2.6.8.5: Profile - Desktop

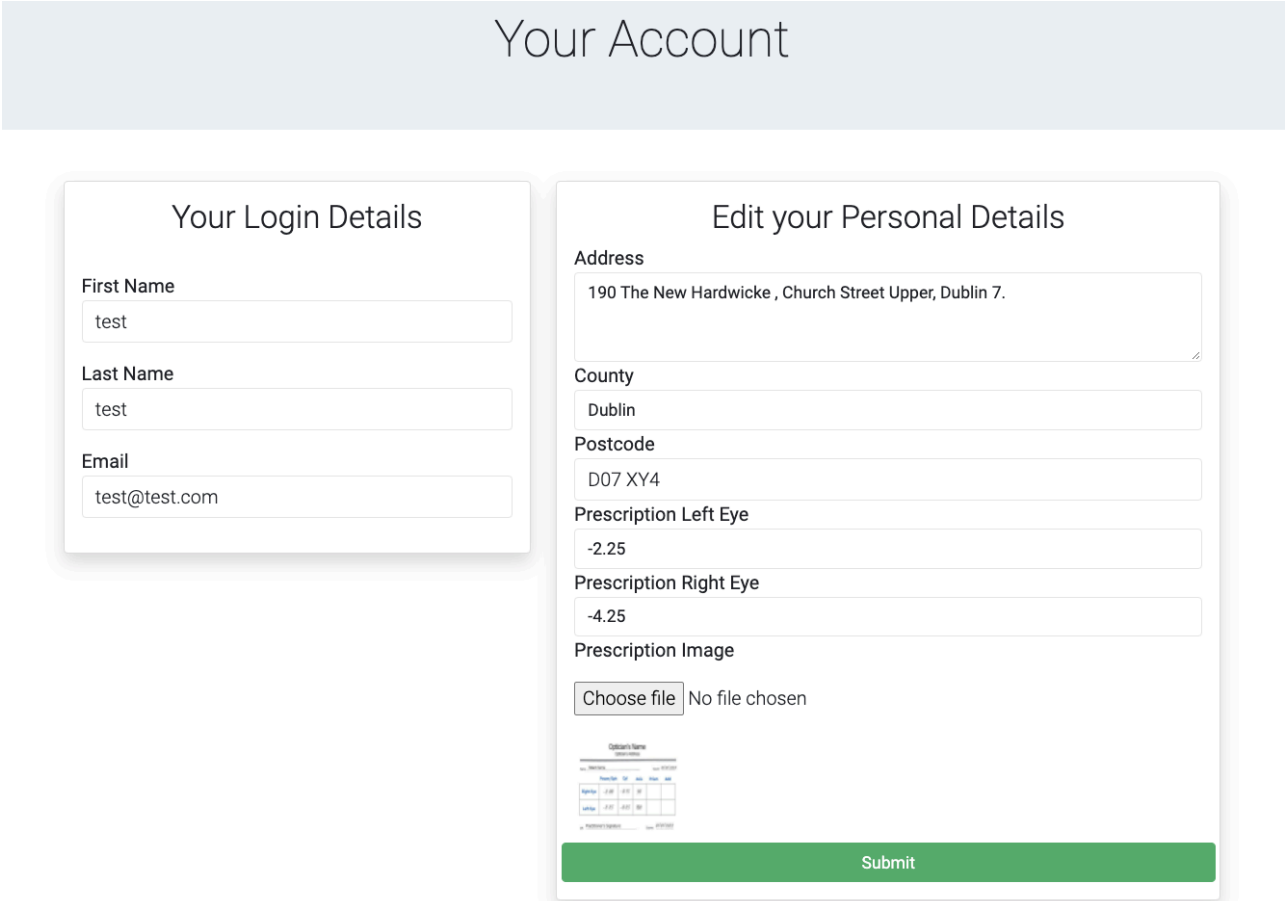

fig2.6.8.6: Add/Edit Profile - Mobile

2.6.9. CMS Homepage

The homepage for the CMS has two shadowed cards that display the options for the administrator. The options are manage the category or product pages. The images shows the mobile view(*fig2.6.9.1*) and the desktop view(*fig2.6.9.2*).

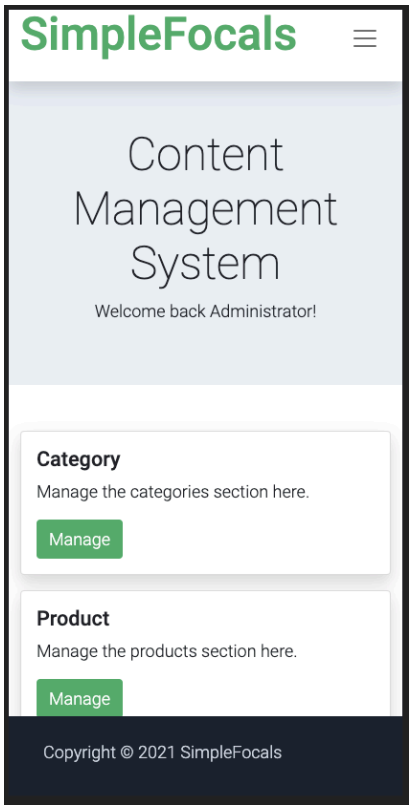

fig2.6.9.1: CMS Homepage - Mobile

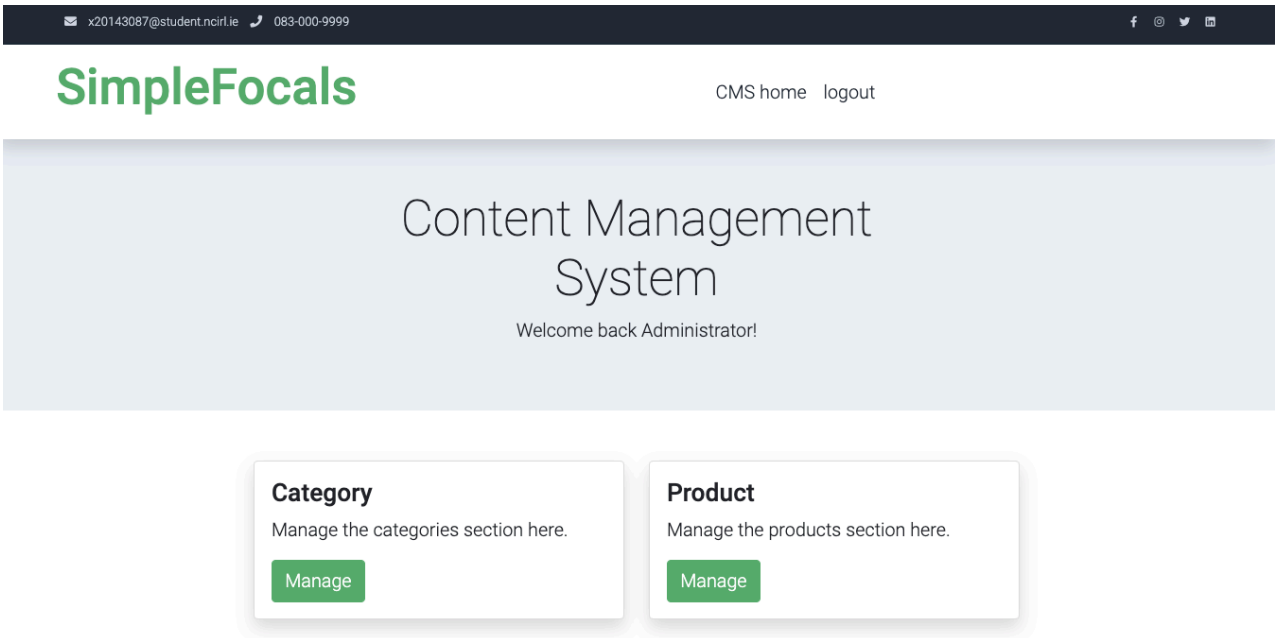

fig2.6.9.2: CMS Homepage - Desktop

2.6.10. CMS Category and Product

The category and product sections of the CMS have two pages each. In the category section one page lists the categories in the system, the other page allows the adding of a category. The product pages are similar. There is one product page that list the products, the other page facilitates the adding of a product. The images show the category(*fig2.6.10.1*) and the add category(*fig2.6.10.2*) page for the mobile view. The images show the category(*fig2.6.10.3*) and the add category(*fig2.6.10.4*) page for the desktop view. The images show the product(*fig2.6.10.1*) and the add product(*fig2.6.10.2*) page for the mobile view. The images product(*fig2.6.10.1*) and the add product(*fig2.6.10.2*) page for the desktop view.

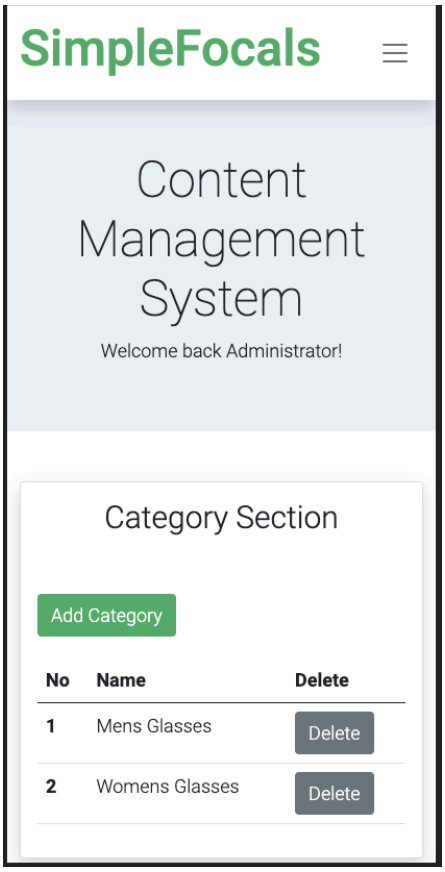

fig2.6.10.1: CMS Category - Mobile

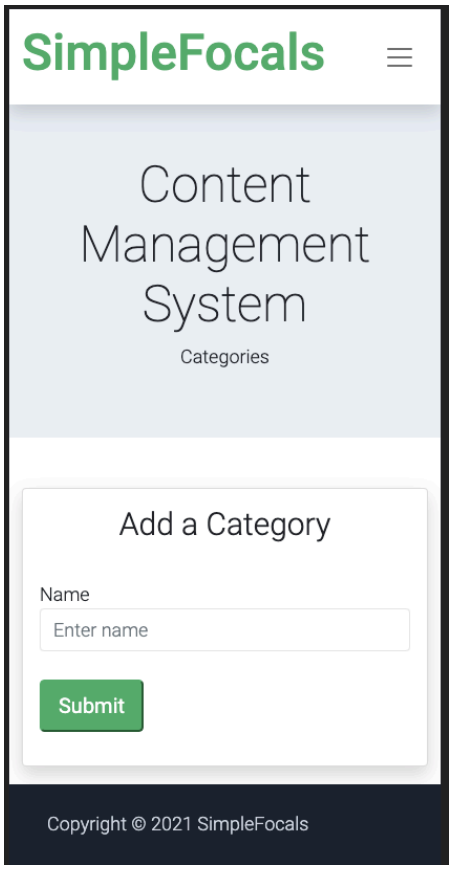

fig2.6.10.2: CMS Add Category - Mobile

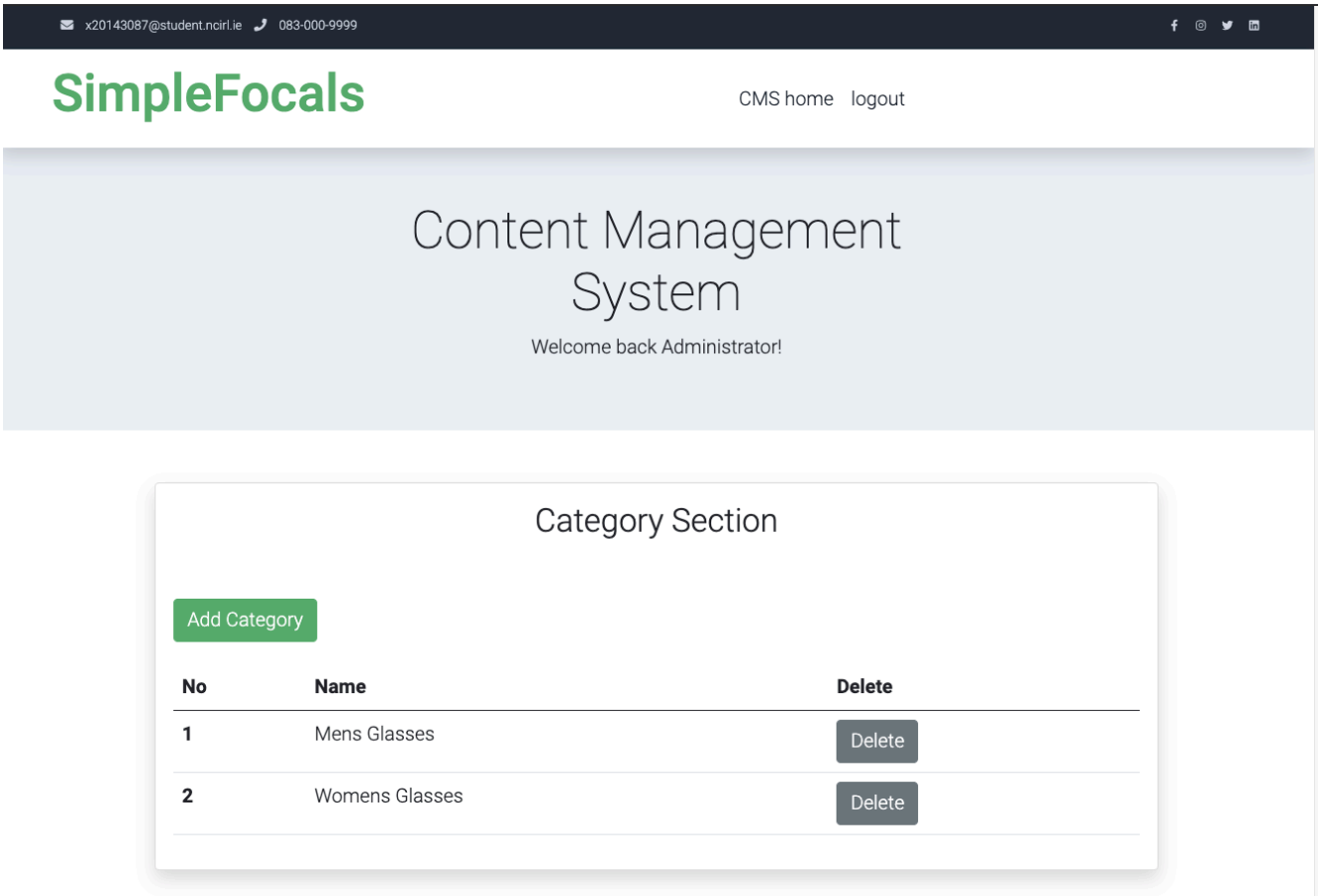

fig2.6.10.3: CMS Category - Desktop

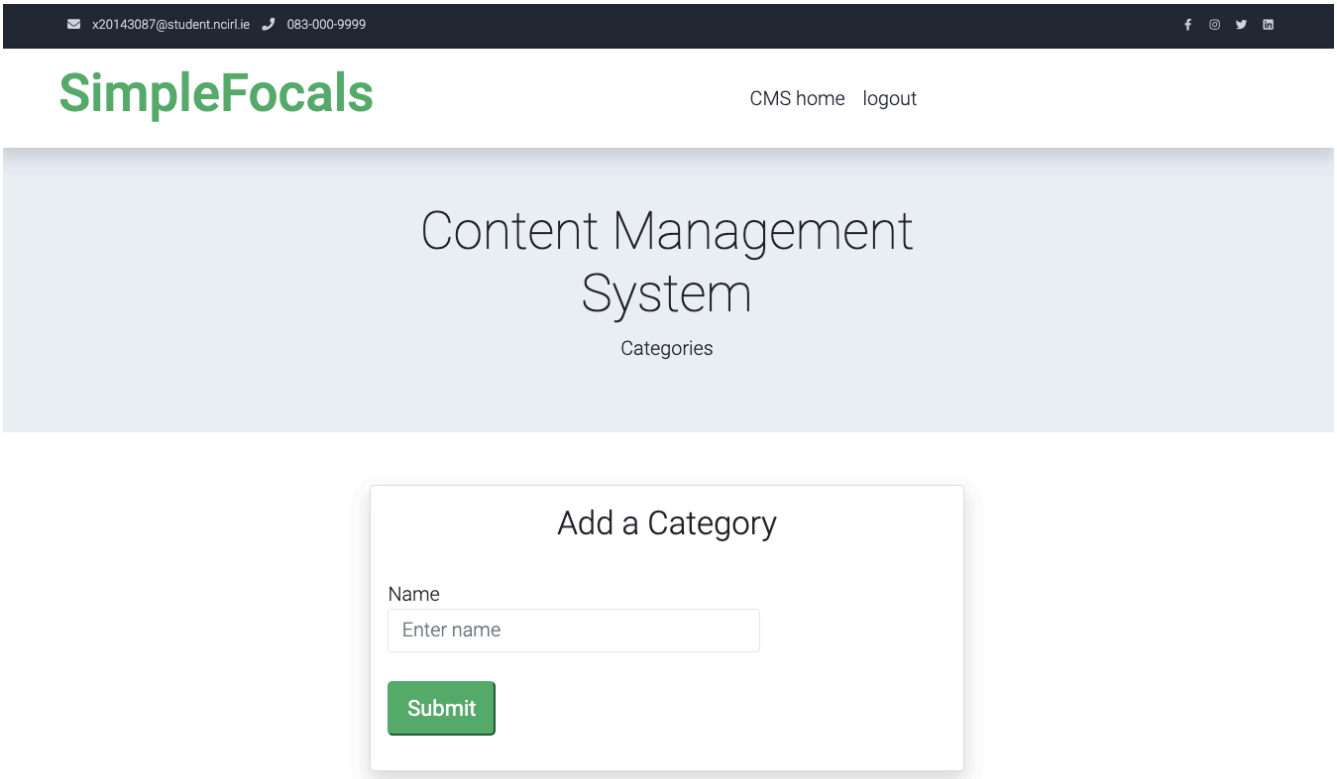

fig2.6.10.4: CMS Add Category - Desktop

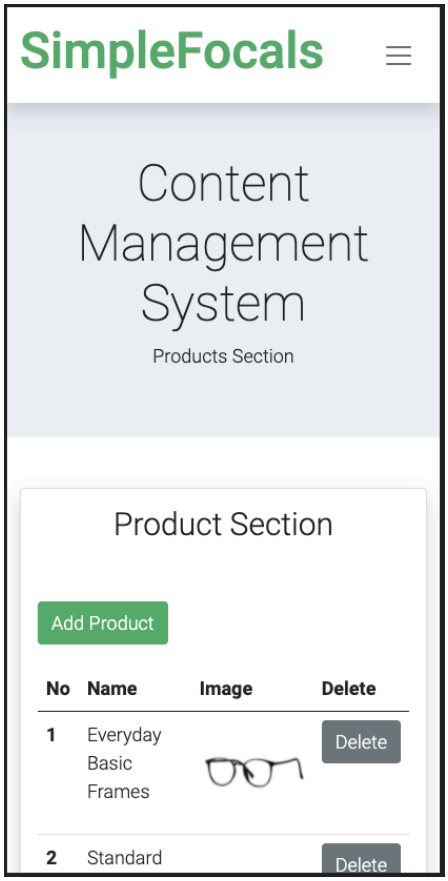

fig2.6.10.5: CMS Product - Mobile

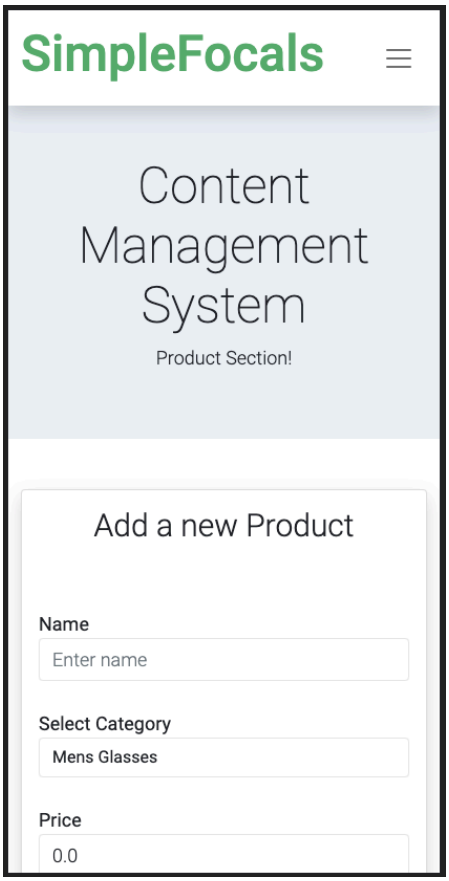

fig2.6.10.6: CMS Add Product - Mobile

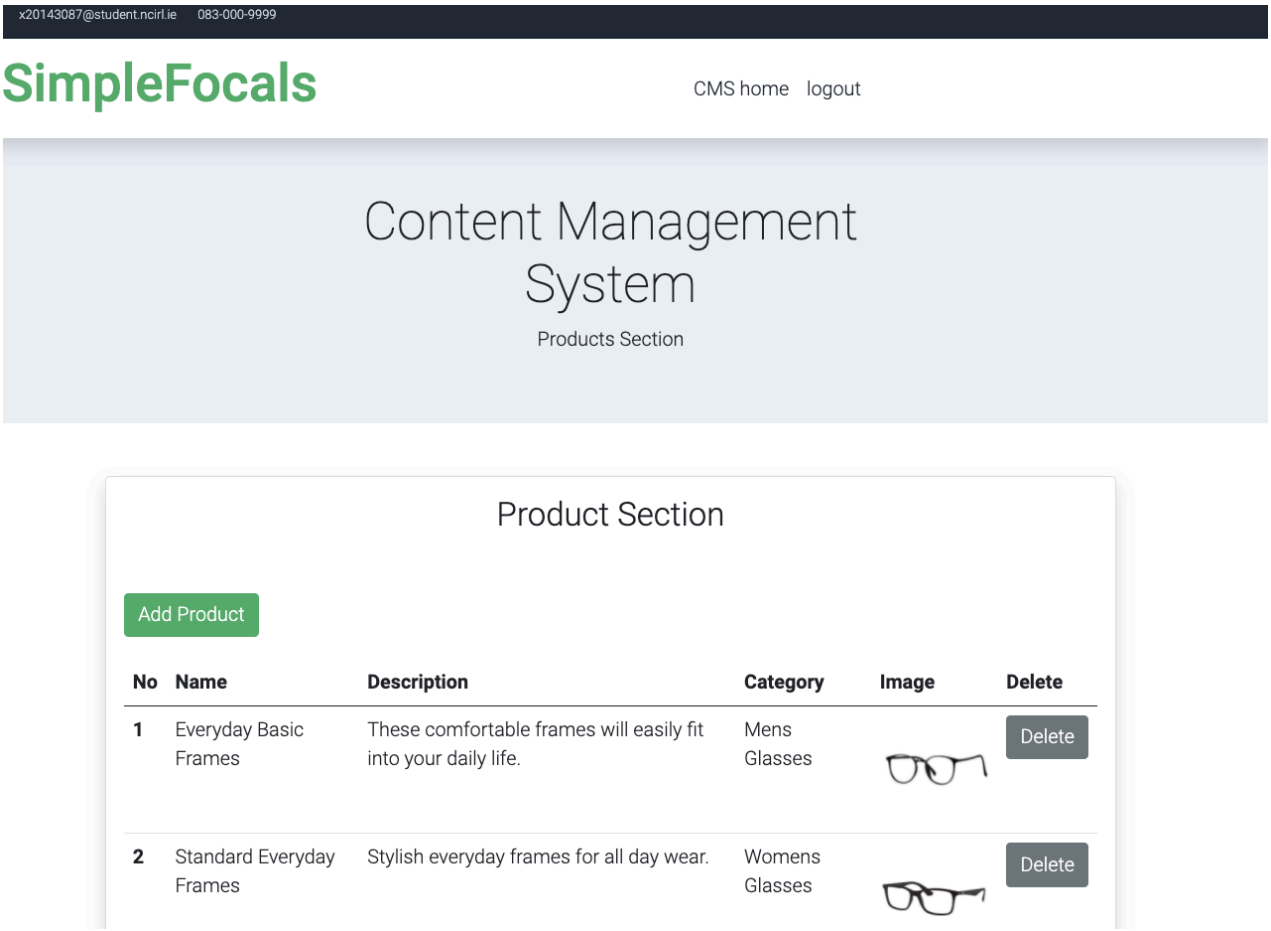

fig2.6.10.7: CMS Product - Desktop

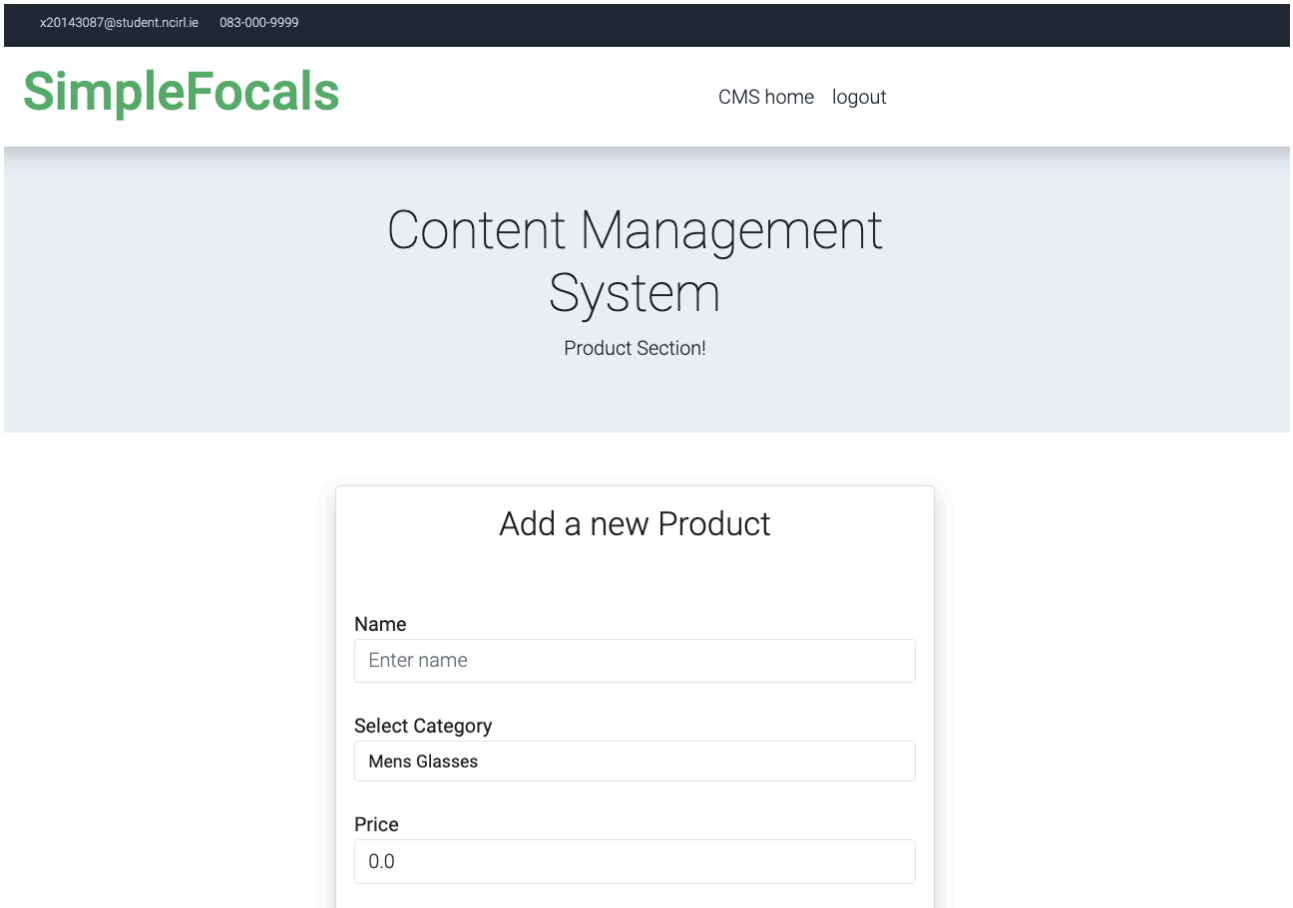

fig2.6.10.8: CMS Add Product - Desktop

2.7. Customer Testing

This section describes what method was conducted for customer testing of the application. I decided to conduct customer acceptance testing for the web application. Acceptance testing is a type of testing that is performed by the client/customer near the end of the software development cycle in order to verify that all the functions and services that are offered by the application are operating as expected. Due to the time and monetary constraints getting a large sample size of participants in the customer testing phase was infeasible. I asked my sister who knew nothing about the application to be a participant in the testing. Before she tested the application I developed a test-case. In the test-case I had fields for the step the participant took in using the application, the expected result of that step, the actual result of the step and whether it pass or failed. When I asked my sister to test the application on her desktop computer, I was busy documenting her usage. I filled out the test-case I had preprepared. The table(*fig2.7.1*) shows the results of the test.

Created By:	Chris	Reviewed By:	Chris	Version	1.0
Testers' Name:	Grace	Date Tested:	19/08/21	Test:(pass/Fail/ Not executed)	Pass
Test Senario:		Register/Login/View product/Add to cart/ Pay			
Step:	**Step Details**	**Expected Results**		**Actual Result**	**Pass/Fail**
1	View Homepage	Application displays and renders all the content.		As Expected	Pass
2	Clicked Login, Register Account	Customers credentials are entered into the fields of the system. The system registers without error. Redirecting back to login screen.		As Expected	Pass
3	Login	Customer enters their login details and the system securely logs them in.		As Expected	Pass
4	Redirect Post-login	The System directs to add customer profile page after login.		As Expected	Pass
5	Add Customer Profile	The system saves the the Customer Profile details and redirects to the profile page.		As Expected	Pass
6	View Shop	When Customer clicks on shop link the application displays the shop items.		As Expected	Pass
7	Filter by Category	When customer clicks on category and the system filters the products for that category.		As Expected	Pass
8	View a Product	When customer clicks on a product the system displays a detailed product page.		As Expected	Pass
9	Increase Quantity of Product	When the customer clicks the quantity input field it increases and decreases accordingly.		As Expected	Pass
10	Add to Cart	When customer has selected a quantity of product and clicks the add to cart option. The items are added to cart. The cart number increases.		As Expected	Pass
11	View Cart	When the customer selects the cart icon the system displays the cart page.		As Expected	Pass
12	Go to Checkout	When the customer clicks go to checkout from the cart page, the system displays the checkout page.		As Expected	Pass
13	Go to payment page	The system displays the payment form with the correct amount to be tendered.		As Expected	Pass
14	Stripe Payment Processing	The customer enters details and Stripe processes payment, system displays payment has been successful.		As Expected	Pass

fig2.7.1: Customer Acceptance Test - Version 1.0

After the testing had finished, I made some minor tweaks to the user interface to make things more clear for the user. The acceptance testing gave me a good insight into a customer that knows nothing about the application approaches using the application. I also took on board some recommendations by the tester that can be used in later revisions of the web application.

2.8. Evaluation

This section describes the evaluation of the project. The application has been consistently evaluated through the project lifecycle. I believe the testing which was documented in ***2.5. Testing*** of this document was a good basis for continual evaluation of the features being built. For end user testing ***2.7. Customer Testing*** of this document describes user acceptance testing that was conducted near the conclusion of the implementation. Results of the user acceptance testing were favourable, all of the main functions of the application were intuitively accessible for a first time user of the e-commerce store.

2.8.1 Google Dev Tools - LightHouse

Lighthouse is an automated tool that is integrated into Chrome browsers that allows for webpage audits. This helps web developers identify which areas of their individual webpages they can improve on. The audit parameters include the performance, accessibility, best practices and SEO. After running the tool on the webpages in the application, the majority of the pages had similar results to the image.

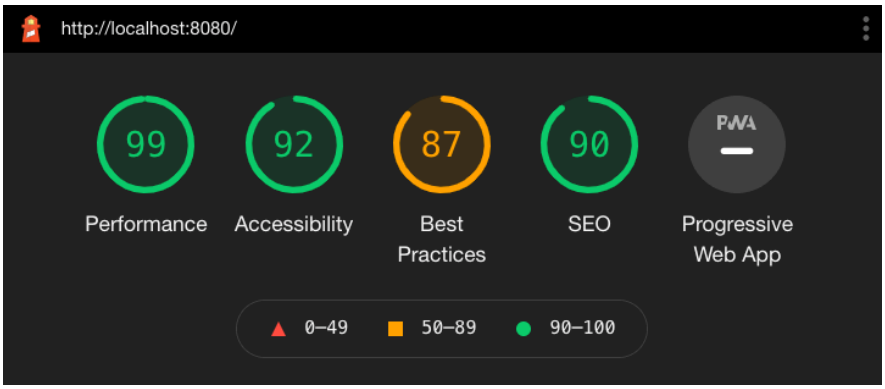

fig2.8.1: Lighthouse score: index.html

The image(*fig2.8.1*) was the score for the index.html homepage. This webpage in the application has three sections and a lot of content. The scores that Lighthouse produced were mostly

in the 90-100 range with one 80-89 score. The image(*fig2.8.2*) shows the score for the home-page of the CMS for the administrator. Similarly to before 3 scores in the 90-100 range and one score in the 80-89 range.

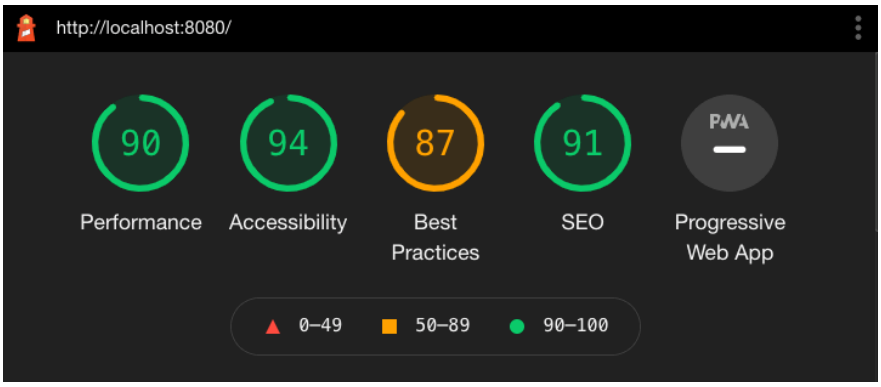

fig2.8.2: Lighthouse score: adminHome.html

These results were largely consistently throughout the whole of the application. Overall the results of the LightHouse evaluation are satisfactory even though there can be a few alterations in some areas to improve results further.

3. Conclusions

This section describes the conclusions that have been gained from the project. I will write this section speaking freely and from my personal experience. I believe this project was a very worth while undertaking. It really made me take the initiative, do my own research and form a structured plan of action when trying to develop a semi complex piece of software. There were a lot of challenges along the way. I think that the jump from second semester to third semester was a big jump. Going from taking classes and being spoon fed the materials to no real starting point and having to fend for ourselves, was a big leap. I was personally worried I wouldn't be able to accomplish what I had set out in my project proposal. Of course there is always the fear of: I don't have enough technical ability, enough knowledge, enough know how to find the information. It was real imposter syndrome. The early documentation, the Requirements Specification document, really helped me arrange my ideas of what needed to be achieved to have a successful project. This began the pathway for me to being able to understand that the goal of

completing this project was realistic. The product design specification then gave me the blueprint on how to implement these ideas. Having some help from my classmates who I formed a study group with, helped me trouble shoot some technical aspects of implementation. Also having the occasion 1-1 with our tutors Liam and Eugene helped clear some issues up. Now finally coming to the conclusion of writing this Technical Document I can say I can be proud of the work I put in this semester. The Journey was long and uncertain at times but we are nearly crossing the finishing line. I'm going to summarise in bullets points the technical things I have learned this semester.

- **Writing technical documents:** During our weekly classes I learned what was expected in technical document writing. Every document I write for this project I feel is an improvement on the last. I still think there is room for improvement though.
- **Starting a Spring Boot MVC application:** Having never developed in this environment before, this was a good starting point for the project. At this stage I was beginning to understand the structure of a MVC web application.
- **Connecting Spring Boot application to MySQL db:** Connecting MySQL db to the Spring Boot application was a good moment. The application was starting to come together.
- **Writing, Reading, Updating items in db:** The eureka moment where, I realised this is how big applications can be developed with modifying data in persistent data stores. I could make the tables, add entries and delete from the application.
- **Developing REST APIs:** I learned that the application could communicate with the front-end though REST API methods, that I could develop.
- **Spring with Thymeleaf integration:** Learned how I could parse JSON objects to send to back-end for processing.
- **DTO use cases:** Learned about dto concept, how the data transfer objects can be used to hide some attributes of an object that we don't want to expose in the application.
- **Using more Bootstrap classes:** I learned more a few more classes in bootstrap and how to use them for formatting.
- **Upload file from front-end to backend:** I learned how to open a file explorer in the browser select a file and have it upload to the system.

- **Deploying a web application:** This was the first application I have deployed. I'm glad for Herokus free tier because it didn't cost any more to host the application.

4. Further development or research

This section describes the further development or research that can be carried at this conclusion of this project. The goal of this project is to develop an application that can form the basis for a commercially viable e-commerce store.

During the deployment of the web application that can the accessed through the URL: *https://serene-ocean-48771.herokuapp.com/* there were some unexpected bugs that could not be resolved before the project submission date. **These errors are only apparent on the deployed app and not an issue when run in a local environment.**

- **CMS Categories Page giving 500 error**: The categories page loads without errors when the application is run in a local environment. It is unusual that this has become a problem when in a deployed environment. More research is required. As a result I will remove a link to the Cateogries.html in the deployed version of the application.
- **The product upload image and upload prescription image doesn't upload to system:**

The file uploader implementations in the application write the files to the directory *src -> main -> resources -> static -> prescriptionImages or productImages.* When the application is deployed on Heroku servers, the image is not uploaded to this directory. More research is required to fix this issue.

The aspects that need to be developed for future versions before commercial deployment are:

- **Email confirmation on registration:** At the current time the registration just redirects to the login page, where the customer can log in with their email and password. Email conformation on registration would limit the number of fake accounts being registered. If when registering the customer has to click an activation link from their email account.
- **More refined GUI:** Custom buttons, instead of default bootstrap. Fix some formatting issues. Rearrange some UI elements. Change the quantity button on the product page.

- **More Pictures of the product on product page:** Currently there is only one picture viewable on the application when a customer clicks into the product. Having the option of viewing more pictures would be desirable feature in the next revision.
- **Pop-up confirmation on adding product to cart:** Currently when a product is added to a cart, the system redirects to the shop page and the cart number is updated in the GUI. A pop-up confirming that the product was added should be implemented.
- **More Functions on the CMS:** Currently the CMS only has add product, delete product, add category, delete category. Extra functionality could be developed here, modify product, add more pictures to product, modify category.
- **Provide Search Functionality of products:** Currently the application can only filter by the categories. Implementation of **s**earch by price - lowest to highest, highest to lowest would be desirable.
- **Create a custom GUI for the payment page:** Currently the application is using the default Stripe payment processing method GUI. Developing SimpleFocals own payment page would be desirable.
- **Integrate other payment methods:** Currently the system is only accepting Stripe payments. Integrating PayPal payment or acceptance of bitcoin is an option.
- **Integrate Google sign-in:** Currently the customer has to register with the system in order to login. With google sign-in integration, if the customer has a google account they can log in using their google credentials and then make a customer profile on the system.

References:

Taylor C, *2021 Specsavers Ireland eyes higher revenues as OCT scan technology rolled out [online] Available at: https://www.irishtimes.com/business/retail-and-services/specsavers-ireland-eyes-higher-revenues-as-oct-scan-technology-rolled-out-1.4119641 [Accessed 5 August 2021].*

Specsavers, *2021 Specsavers Ireland All Galsses [online] Available at: https://www.specsavers.ie/glasses/all-glasses?esp_product_content_sort=price&esp_product_content_order=asc [Accessed 5 August 2021].*

VisionExpress, *2021 VisionExpress Our Story [online] Available at: https://www.visionexpress.ie/more-at-ve/about-us [Accessed 5 August 2021].*

Iris Oifigiúil, *(2015) S.I. No. 455/2015 - Optical Registration Board (Regulation and Control of Prescribing and Dispensing of Prescriptions and Sale of Spectacles) Bye Law 2015 [online] Available at: http://www.irishstatutebook.ie/eli/2015/si/455/made/en/print [Accessed 8 June 2021].*

Cassiomolin, *2016 Why you should use DTOs in your REST API [online] Available at: https://cassiomolin.com/2016/03/23/why-you-should-use-dtos-in-your-rest-api/ [Accessed 15 August 2021].*

6. Appendix

6.1. Credentials

Web Application

The credentials for the web application:

Administrator:

Username: christobelchan@gmail.com

Password: 12345678

Customer (with Customer Profile):

Username: test1@test.com

Password: 87654321

Customer(without Customer Profile):

Username: test4@test.com

Password: 87654321

The system allows registering new customers. So when testing, the user can just register an account instead of using existing test customer's account.

Database

If restoring the SQL dump file and loading the database please use these credentials.

SimpleFocals1:

Username: root

Password: rootroot

6.2. Requirements Specification

Name: Christopher Chan

Student Number: 20143087

Class: HDSDEV_SEP (Higher Diploma in Science of Computing)

Module: Final Project

Requirements Specification:

SimpleFocals

National College of Ireland

Requirements Specification (RS)

Document Control

Revision History

Date	Version	Scope of Activity	Prepared	Reviewed	Approved
25/06/21	1	Create	AB	X	X
20/08/21	1.1	Edit			

Distribution List

Name	Title	Version
Liam McCabe	Lecturer	1

Related Documents

Title	Comments

Table of Contents

1. Introduction 4

1.1 Purpose. 4

1.2 Project Scope 4

1.3 Definitions, Acronyms, and Abbreviations. 5

2. User Requirements Definition 6

3. Requirements Specification. 6

3.1. Functional Requirements. 6

3.1.1. Top Level Use Case Diagram 7

3.1.2. Requirement 1: View Product 7

3.1.3. Requirement 2: Register Customer. 9

3.1.4. Requirement 3: Login. 10

3.1.5. Requirement 4: Add to Cart 12

3.1.6. Requirement 5: Upload Prescription. 13

3.1.7. Requirement 6: Make Payment 14

3.1.8. Requirement 7: Add Product 17

3.2. Non-Functional Requirements. 17

3.2.1. Performance/Response time requirement. 17

3.2.2. Availability requirement. 17

3.2.3. Recover requirement. 17

3.2.4. Robustness requirement. 17

3.2.5. Security requirement. 17

3.2.6. Reliability requirement. 18

3.2.7. Maintainability requirement 18

3.2.8. Portability requirement. 18

3.2.9. Extendibility requirement. 18

4. Interface Requirements 18

4.1. Graphical User Interface (GUI) 18

4.2. Application Programming Interface (API) 23

5. System Architecture. 23

6. System Evolution. 23

1. Introduction

1.1 Purpose

The purpose of this document is to outline the requirements for the development of SimpleFocals E-commerce web application. The web application will be the entry point where customers can browse SimpleFocals range of eyewear, add items to a cart and complete their purchases. The intended customers for SimpleFocals are people who have miopic(short-sighted) prescriptions and are looking for a cost effective alternative to the eyewear sold in the Irish market.

1.2 Project Scope

- The web application will facilitate, a customer from registering with the system until the end point of purchasing items in SimpleFocals range of eyewear.
- The web application will facilitate an administrator's access to the mange the application. The administrators should be able to add products in SimpleFocals range.

Out of Scope:

- The web application will not support numerous types of payment processing.
- Adding tax to the products at checkout(later revisions this will be added).
- In the initial version of the web application will not have reviews from customers about purchased products although future updates will facilitate this functionality.
- Automatic E-mail for acceptance of registration.
- Javascript modal front-end notifications. E.g adding to cart. Payment verification.

Motivation for this Project:

The motivation for this project is to provide the Irish prescription glass wearing consumer a cost effective alternative to existing eyewear vendors.

Project Objectives:

The project aims to deliver these functions for users of the web application.

Customers will be able to:

- Browse SimpleFocals range of eyewear.
- Register with the system.

Registered-Customers will be able to:

- Upload their eye prescription to the system
- Add product to a cart.
- Purchase a product.

Administrators will be able to :

- Add products to the shop.

Project Expectations:

At the completion of this project, the web application will form the basis for a fully functioning E-Commerce web application although the system will not be ready for commercial use initially.

Project Suppositions and Restrictions:

Receiving feedback from market research during the planning stage can be limited as we have a small window of time before the project has to move on to the development stage. Therefore the sample size of the market research could be smaller than expected.

Project Risks and Contingency Plan:

The main risk to the project is time constraints for implementing all the features and completing all the documentation required to a high standard. In order to accomplish this, during the project lifecycle, some features are open to modification in order to meet the required milestones and deadlines.

1.3 Definitions, Acronyms and Abbreviations

User:	The user of the system refers to the customer and the administrator
Customer:	Registered customer and non-registered customer
Administrator:	A user that has special privileges to modify products on the system
E-Commerce:	Electronic Commerce is the medium of selling of goods/services on the internet.
URL:	Uniform Resource Locator is the browsers mechanism to retrieve a published resource on the internet.

PDF: A standard file format for digital documents, originally developed by Adobe.

JPEG: A compressed file format for images.

CMS: Content Management System that administrators use to edit content in an application

2. User Requirements Definition

This section outlines the requirements a user must have in order to use the web application.

- **Internet Browser:** The web application can be interfaced using the browser.
- **Internet Connection:** The user will require access to internet connectivity in order to interact with the web application through the browser
- **Computing Device:** The computing device (computer/laptop, tablet or smartphone) should have access to an active internet connection and internet browser.

3. Requirements Specification

This section outlines all of the functional requirements that the web application should accomplish and also describe the main details of the system and how the Users interact with the system. Non-functional requirements shall also be investigated.

3.1 Functional Requirements

The functional requirements describes what the system **must be able to accomplish**. The requirements are ranked in terms of priority (*1 = highest priority, 7 = lowest priority*).

Ranked Requirements:

1. Customer is able view SimpleFocals range of eyewear.
2. Non-registered customer is able to register with the system.
3. User is able to login to the system.
4. Registered customer is able add to item/s to cart.
5. Registered customer is able to upload a digital copy of their eye prescription.

6. Registered customer's payment is able to be processed.
7. Administrator is able to add products.

3.1.1 Top Level Use Case Diagram

The use case diagram outlines the actors and how they interact with the system. SimpleFocals System is described above(*fig3.1*).

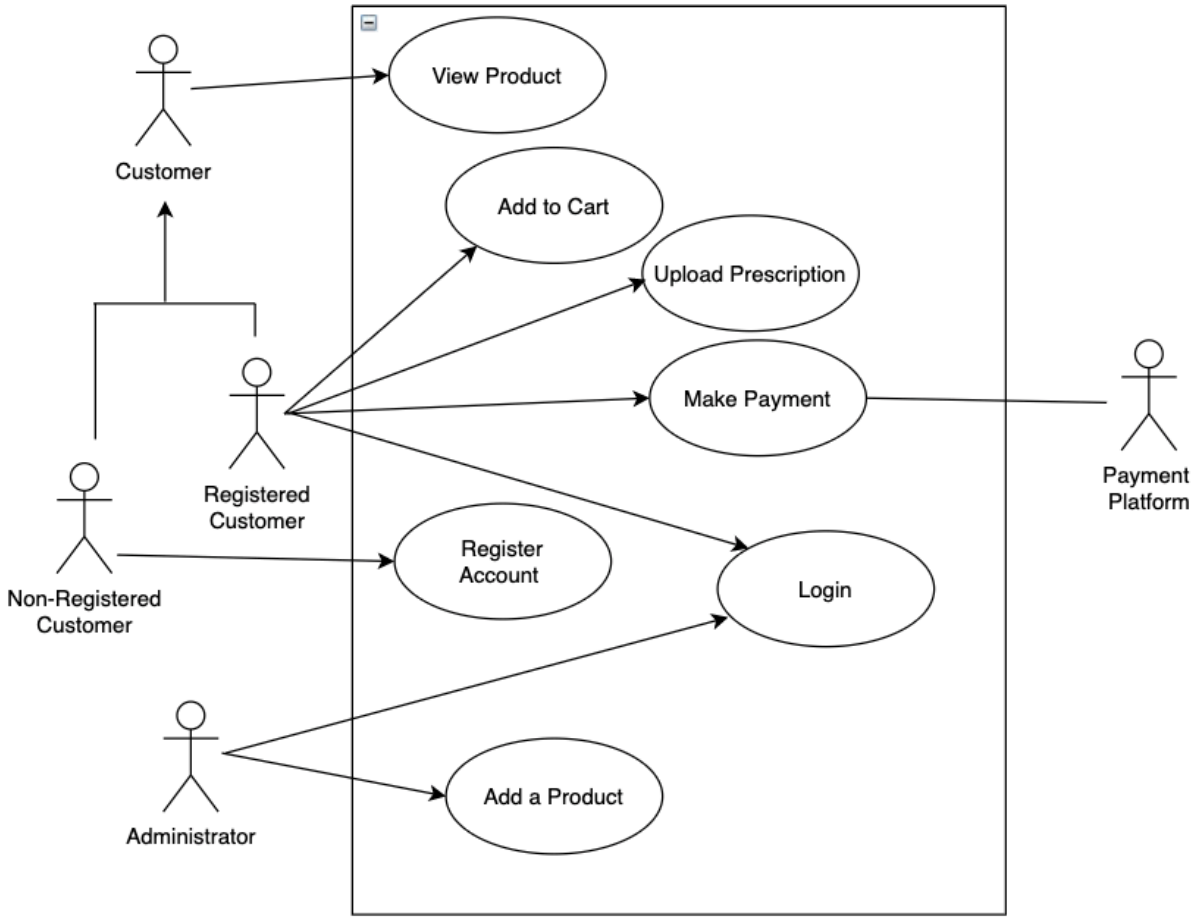

fig3.1: Top Level Use-Case Diagram for SimpleFocals System

3.1.2. Requirement 1: View Product.

This is requirement 1 of the system, the customer should be able to view SimpleFocals range of eyewear. The use-case description outlines how the actors interact with the system(*fig3.2*).

Name	**View Product**
Description	The customer views SimpleFocals range of different eyewear products.
Priority	This use-case is of high **priority 1** because it is the starting point for potential sales in the E-commerce application.
Precondition	None
Flow Description	
Activation	This use case can be activated on the navigation bar of the web application. The customer clicks on the shop tab on the navigation bar.
Main Flow	1. The customer clicks on the shop tab on the navigation bar. 2. The system displays categories on the left of the screen. 3. The customer selects the category they would like to view. 4. The system displays the all products for that category. 5. The customer selects a specific set of glasses to view. 6. The system displays a detailed view of the glasses.
Alternate Flow	
Title	**Description**
Customer enters direct URL	A1. The customer enters into the web browser address field: *{the web. site URL}/shop* A2. The main flow from 2 is continued.
Exceptional Flow	
Title	**Description**
Post-Condition	
Title	**Description**
Termination	When the customer leaves the shop page the use-case has ended

fig3.2: use-case description for requirement 1

3.1.3. Requirement 2: Register Customer.

This is requirement 2 of the system, a non-registered customer should be able to register with the system. The use-case description outlines how the actors interact with the system(*fig3.3*).

Name	**Register Customer**
Description	A non-registered customer registers with the system.
Priority	This use-case is of **priority 2** because the system requires the customer to be registered before making any purchases.
Precondition	The customer has not previously registered with the system.
Flow Description	
Activation	This use case is activated when the customer navigates to the login page. The customer selects the register option.
Main Flow	1. The customer selects the option register. 2. The system displays fields that have to be entered by the Customer(First Name, Last Name, Email Address and Password) 3. The customer enters the required fields and then selects the click here to register option. 4. The system redirects to the login page if the registration was successful.
Alternate Flow	
Title	**Description**
Email previously registered	A1. The customer tries to register an email address that has been previously registered to the system. A2. The system displays the email address already exists on the system. The system indicates to change the email address. A3. The system continues from main flow 3.
Exceptional Flow	
Title	**Description**
Post-Condition	

Title	Description
Termination	The system displays redirects to the login page.

fig3.3: use-case description for requirement 2

3.1.4. Requirement 3: Login.

This is requirement 3 of the system, a user is able to log into the system. A user can be a registered customer or the system administrator. The use-case description outlines how the actors interact with the system(*fig3.4*)(*fig3.5*).

Name	**Login (Customer)**
Description	A registered customer logins into the system.
Priority	This use-case is of **priority 3** because the system requires a login function for customer and administrator.
Precondition	The customer has previous registered on the system.
Flow Description	
Activation	On the navigation bar at the homepage of the application there is an option for login. The customer selects the login option.
Main Flow	1. The system displays the input fields Email Address and Password. 2. The customer enters the required fields and then selects the click to login option. 3. The system signs-in the customer.
Alternate Flow	
Title	**Description**
User fails authentication	A1. The customer enters an email address incorrectly or enters a password that doesn't correspond to a given email address A2. The system displays invalid username or password. A3. The customer corrects one of the fields giving the error A4. The system continues from main flow 2.

Exceptional Flow	
Title	**Description**
Post-Condition	
Title	**Description**
Termination	The customer is redirected to a homepage for their account.

fig3.4: use-case description for requirement 3(registered-customer)

Name	**Login (Administrator)**
Description	An administrator logins into the system.
Priority	This use-case is of **priority 3** because the system requires a login function administrator to access the CMS.
Precondition	The administrator has an account on the system.
Flow Description	
Activation	On the navigation bar at the homepage of the application there is an option for login. The administrator selects the login option.
Main Flow	1. The system displays the input fields Email Address and Password. 2. The administrator enters the required fields and then confirms the selects the click to login option. 3. The system signs-in the administrator.
Alternate Flow	
Title	**Description**
User fails authentication	A1. The administrator enters an email address incorrectly or enters a password that doesn't correspond to a given email address A2. The system displays invalid username or password. A3. The administrator corrects one of the fields giving the error. A4. The system continues from main flow 2.

Exceptional Flow	
Title	**Description**
Post-Condition	
Title	**Description**
Termination	The system displays the CMS/administrator homepage.

fig3.5: use-case description for requirement 3 (administrator)

3.1.5. Requirement 4: Add to Cart.

This is requirement 4 of the system, a registered customer is able to add item/s to a cart. The use-case description outlines how the actors interact with the system(*fig3.6*).

Name	**Add to Cart**
Description	The registered customer adds products to the cart
Priority	This use-case is of **priority 4** because adding to the cart is the next step in the purchase process after signing into the system.
Precondition	The customer has registered and logged into the system.
Flow Description	
Activation	This use case is activated on the view product page of the web application. The customer clicks add to cart.
Main Flow	1. The customer selects the quantity of the product. 2. The system displays the updated quantity of the product. 3. The customer clicks on add to cart option.
Alternate Flow	
Title	**Description**

Exceptional Flow	
Title	**Description**
Post-Condition	
Title	**Description**
Termination	The cart icon number increases and the use-case terminates.

fig3.6: use-case description for registered-customer requirement 4

3.1.6. Requirement 5: Upload Prescription.

This is requirement 5 of the system, a registered customer is able to upload a digital copy their eye prescription. The use-case description outlines how the actors interact with the system(*fig3.7*).

Name	**Upload Prescription**
Description	The registered customer uploads a digital copy their eye prescription to the system.
Priority	This use-case is of **priority 5** because it is next in the purchase flow of events.
Precondition	1. The customer has registered with the system. 2. The customer logins into the system.
Flow Description	
Activation	This use case is activated after the customer has registered and logins for the first time to the system. The system displays the customer profile page with “add more details to your profile button” Once this button is selected, the customer should fill out including details of address, county, postcode and their prescription details and select submit.

Main Flow	1. The system displays the customer's profile page. 2. The customer selects the option to add more details to your profile. 3. The system displays many fields that should be filled out including address, county, postcode, prescription left eye, prescription right eye and choose file option to upload prescription. 4. The customer fills out the fields and selects the choose file option. 5. The system displays a new file browser window to the customer where they can search the location of the prescription locally on their computing device. 6. The customer selects the prescription file with compatible format. 7. The customer selects the upload file option.
Alternate Flow	
Title	**Description**
Exceptional Flow	
Title	**Description**
Post-Condition	
Title	**Description**
Termination	The system redirects to display the profile page showing the prescrip-tion file is now on the system.

fig3.7: use-case description for registered-customer requirement 5

3.1.7. Requirement 6: Make Payment.

This is requirement 6 of the system, a registered customer's payment is able to be processed. The use-case description outlines how the actors interact with the system(*fig3.8*).

Name	**Make Payment**
Description	The registered customer is at checkout and pays for the products in the cart.
Priority	This use-case is of **priority 6** because this is the final stage of the purchase process for the customer.
Precondition	1. The customer has registered and logged into the system. 2. The customer has uploaded a valid prescription in a compatible format (pdf or jpeg/png). 3. The customer has added product/s to the cart. 4. The customer has navigated to the cart items.
Flow Description	
Activation	This use case is started when the customer selects the pay now option on the checkout page of the web application.
Main Flow	1. The customer selects the pay now option from the checkout page. 2. The system will load a payment page. 3. The customer will enter their payment details(credit/debit card number, card expiration date and card verification value) and 4. The customer will select confirm transaction. 5. The system will display the payment has been successful.
Alternate Flow	
Title	**Description**
Exceptional Flow	
Title	**Description**
Invalid payment details	A1. If the customer enters invalid card details the system displays the payment is not accepted and indicates the error. A2. The customer is then promoted to correct the payment information entered.
Post-Condition	
Title	**Description**

Termination	The customer will be notified that the payment was successful and the system displays a link where the customer can return to the shop.

fig3.8: use-case description for requirement 6

3.1.8. Requirement 7: Add a Product

This is requirement 7 of the system, an administrator is able to add a product to the system. The use-case description outlines how the actors interact with the system(*fig3.9*).

Name	**Add a Product**
Description	An administrator adds a product using the CMS on the application.
Priority	This use-case is of **priority 7** because the system requires the Administrator the functionality to add products.
Precondition	The administrator has an account on the system.
Flow Description	
Activation	This use case is started when administrator selects the Add Product from the products page on the CMS.
Main Flow	1. The system displays the input fields Name, Category, Price, Product description and Product Image(opens file explorer upload file locally). 2. The administrator inputs all the required fields. 3. The administrator selects the submit button. 4. The system displays the product was added successfully.
Alternate Flow	
Title	**Description**
Exceptional Flow	
Title	**Description**

Post-Condition	
Title	**Description**
Termination	The system redirects back to the products page and displays the added product to the list of products in the shop.

fig3.9: use-case description for requirement 7

3.2. Non-Functional Requirements

This section describes the non-functional requirements of the project. This section will outline the performance/response time, availability, recovery, robustness, security, reliability, maintainability, portability and extendibility requirements.

3.2.1. Performance/Response time requirement

The web application when deployed should handle 100 users at a given time. The following conditions are expected to be satisfied:

- Login shall not take longer than 3 seconds.
- Loading of pages shall not take longer than 5 seconds

3.2.2. Availability requirement

The web application will be deployed to a cloud service before the deadline for final project submission. The selected cloud service will satisfy the performance and response time constraints. The availability for the user of the system requires a computing device that has a browser connected to internet.

3.2.3. Recovery requirement

The system should be able to recover where possible and allow the user to continue to access system functions.

3.2.4. Robustness requirement

The deployed web application should be able to handle errors appropriately. The system should be able to recover where possible and not affect availability.

3.2.5. Security requirement

The deployed web application will not be ready for commercial release when the project is completed. The project is a proof of concept implementation but will follow secure implementations for login and registration. Payment processing will be securely handled by a 3rd party.

3.2.6. Reliability requirement

The cloud service that is chosen to host the web application should be able to satisfy the availability and Performance/Response time requirement.

3.2.7. Maintainability requirement

The web application should be easy to modify and extend the code base in preparation for commercial implementation.

3.2.8. Portability requirement

The web application should be accessible on different device form factors(computer, smartphone and tablet devices).

3.2.9. Extendibility requirement

The web application being developed for this project is a proof of concept implementation. The project should be extendible to a commercially ready application with more development.

4. Interface requirement

This section describes the interface requirements for the project. The graphical user interface (GUI) and any application programming interfaces(APIs) will discussed.

4.1. Graphical User Interface (GUI)

I will take a mobile first design to create the graphical user interface. Prototyping websites on a smaller screen first helps to ensure that user experience is consistent regardless of device. I will illustrate the the GUI using simple wireframes for each function of the system. In the use-case, view product, the interface will look similar to the image below(*fig4.1*). The user will click the products for a more detailed view of the product. The use-case add to cart is activated from the image(*fig4.2*). The registered customer will click the add to cart button. The use-case upload prescription is activated from the image(*fig4.3*). The use-case make a payment is activated from the image(*fig4.4*). The use-case register customer is activated from image(*fig4.5*). The use-case

for a customer or administrator to login to the system is activated from image(*fig4.6*). The use-case for an administrator to add a product is activated from the image(*fig4.7*).

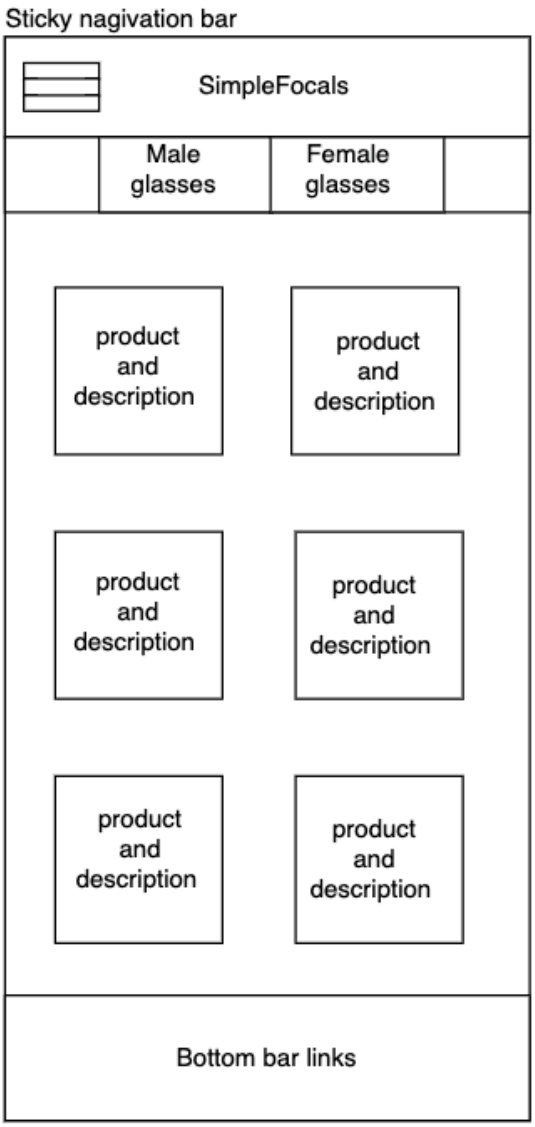

fig4.1: Product Page - view product

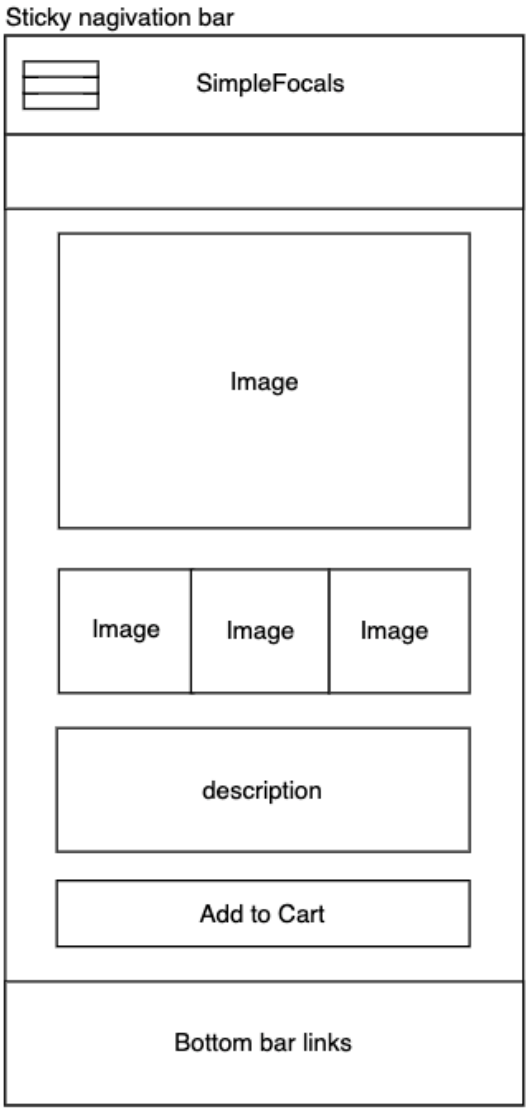

fig4.2: Detailed Product Page - add to cart

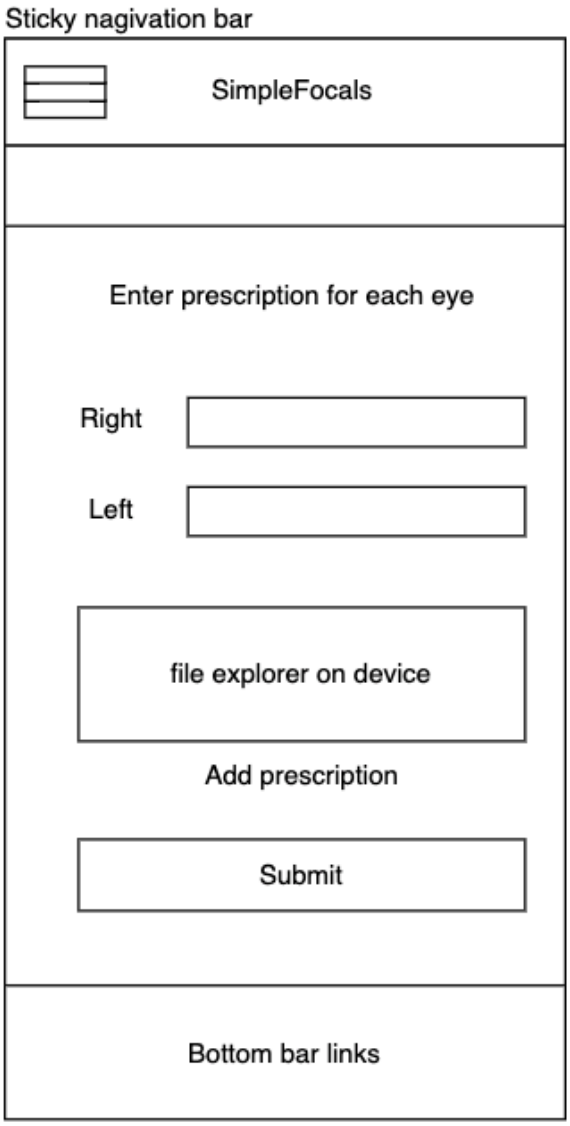

fig4.3: Upload Prescription page - upload prescription

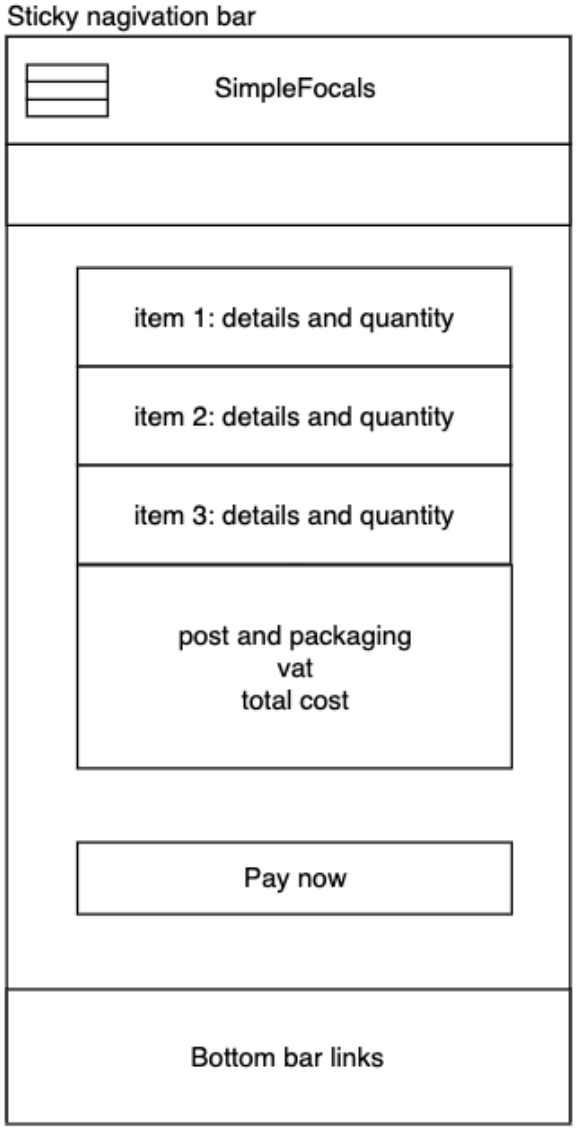

fig4.4: Payment page - make a payment

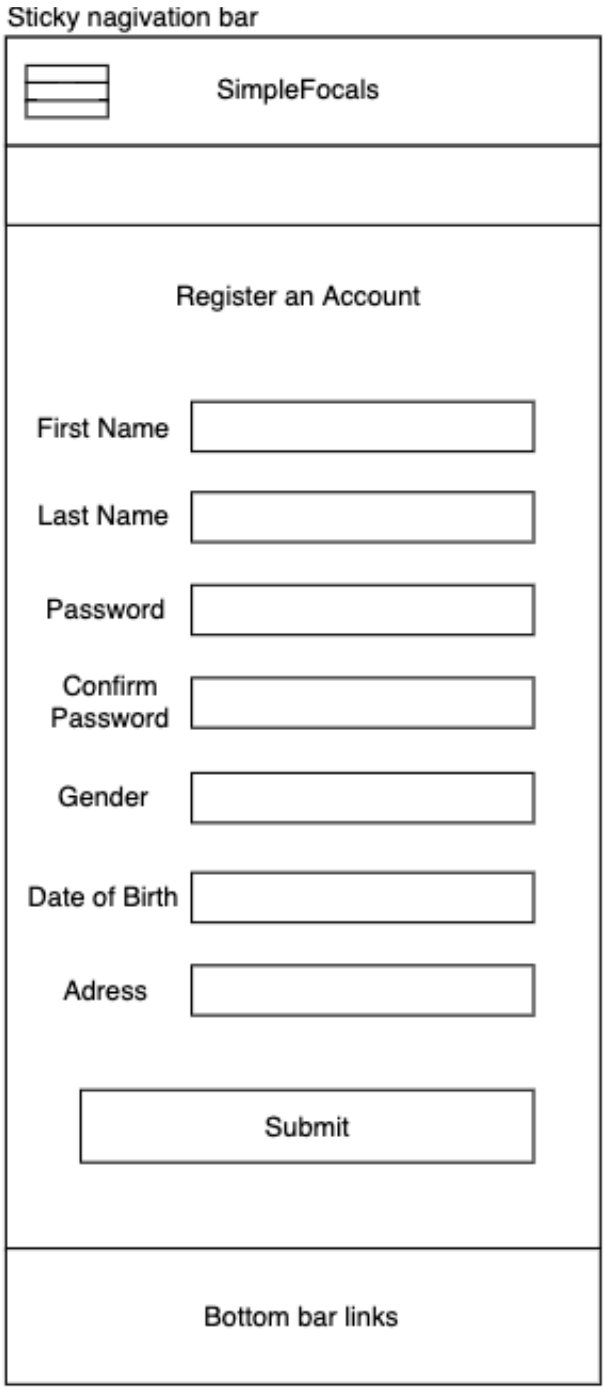

fig4.5: Registration page - register customer

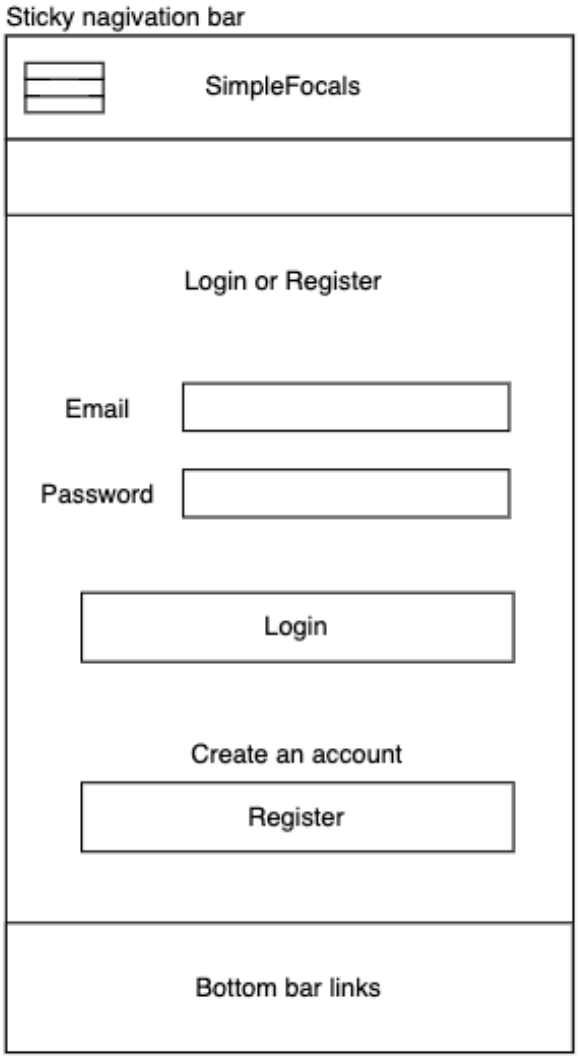

fig4.6: Login page - log in to system

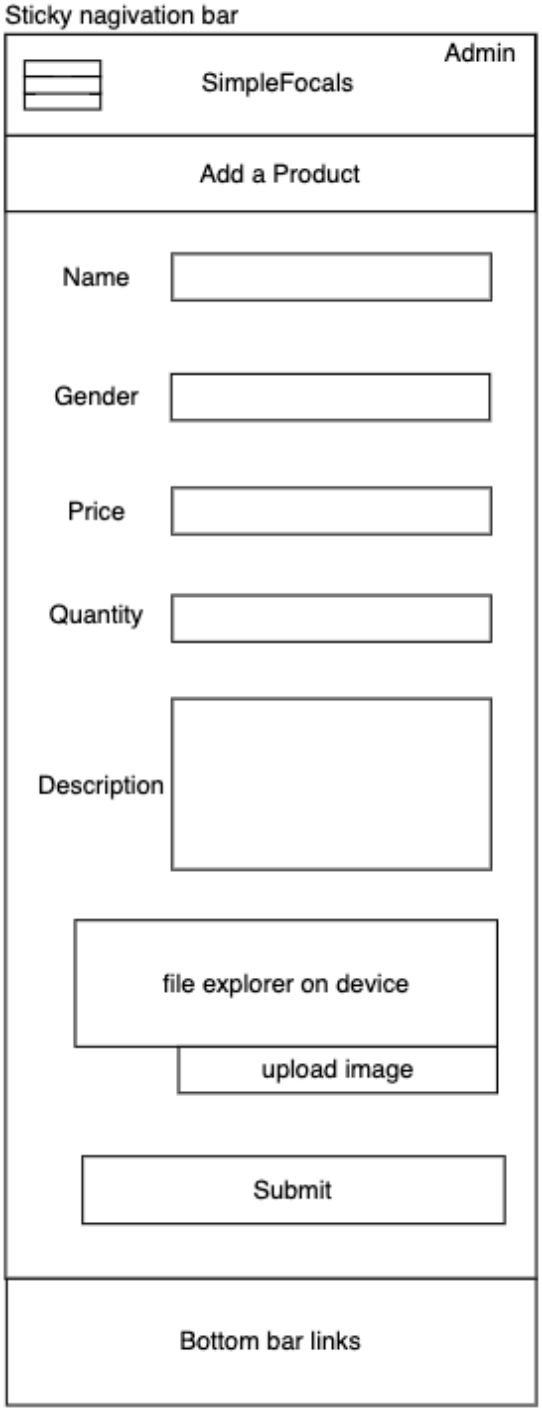

fig4.7: Add product admin page - add product

4.2. Application Programming Interface (API)

The payment processing will be handled by an external 3rd party service. I am currently researching integrating the Stripe payment API for use in the web application. The use of external payment API means that no sensitive card details are stored within the web application. Those details will be discretely and securely processed by the trusted 3rd party. The web application will only store a record of the transaction in the system.

5. System Architecture

The class diagram(*fig5.1*) is the initial design structure for the web application. During future revisions this structure is open to change.

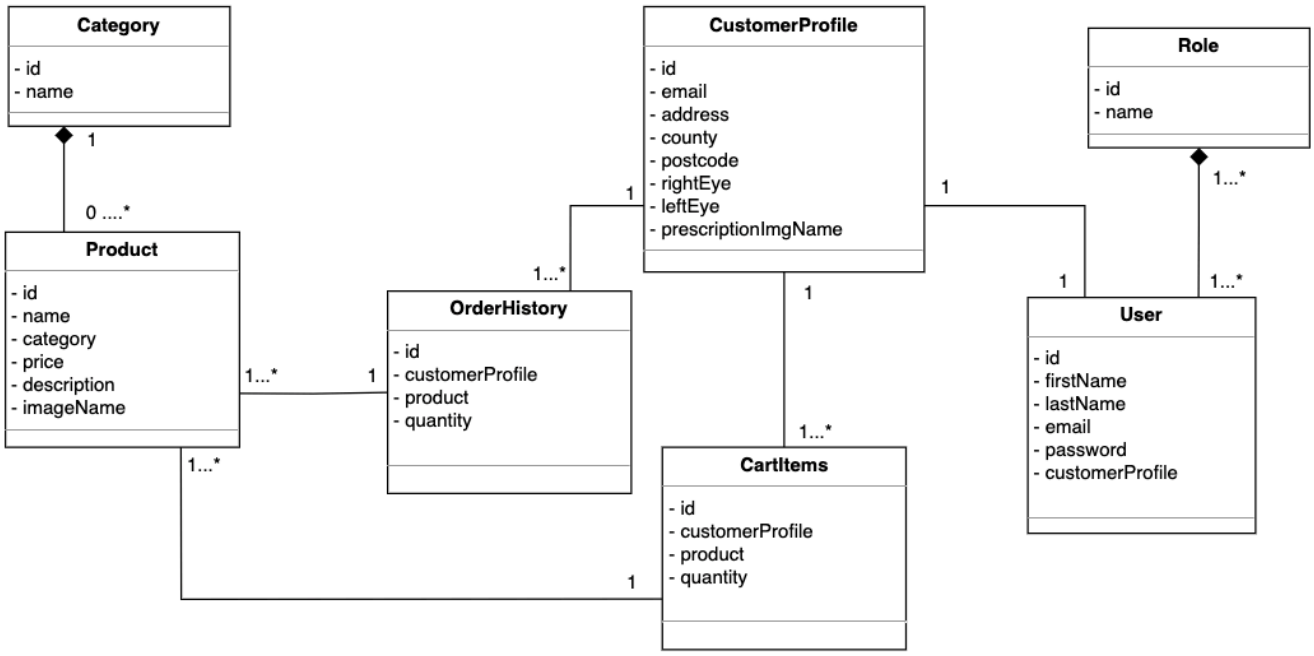

fig5.1: Initial Conceptual Class Diagram for SimpleFocals

6. System Evolution

Future revisions of the web application could bring more features.

1. Adding Customer reviews (star ratings for product and service).
2. Increase functionality on Administrator dashboard (content search, manage reviews)
3. Add subscription model for purchasing eyewear.

6.3. Product Design Specification

Name: Christopher Chan

Student Number: 20143087

Class: HDSDEV_SEP (Higher Diploma in Science of Computing)

Module: Final Project

Product Design Specification:

SimpleFocals

National College of Ireland

Product Design Specification (RS)

Document Control

Revision History

Date	Version	Scope of Activity	Prepared	Reviewed	Approved
10/07/21	1	Create	AB	X	X
20/08/21	1.1	Edit			

Distribution List

Name	Title	Version
Liam McCabe	Lecturer	1
Liam McCabe	Lecturer	2

Related Documents

Title	Comments

Table of Contents

1. Introduction. 4

1.1. Purpose of The Product Design Specification Document 4

2. General Overview and Design Guidelines/Approach. 4

2.1. Assumptions/Constraints/Standards 4

3. Architecture Design. 5

3.1. Logical view. 5

3.2. Hardware Architecture 7

3.3. Software Architecture 8

3.3.1. Front-end. 8

3.3.2. Back-end. 9

3.4. Security Architecture. 10

3.5. Communications Architecture. 11

3.6. Performance. 13

4. System Design. 13

4.1. Use Cases. 13

4.2. Database Design. 13

4.2.1. Core entities of the business 13

4.2.2. Relationships and cardinalities 16

4.2.3. No Relationship Table. 17

4.2.4. Entity Relationship Diagram 17

4.2.5. Logical tables 17

4.2.6. Database service and Spring Boot integration 19

4.2.7.Class/Relation Implementation 19

4.3. Data Conversions 22

4.3.1. Back-End. 22

4.3.2 Front-End 22

4.4. Application Programming Interfaces 22

4.5. User Interface Design. 23

4.6. Performance. 23

4.6.1. Why use java for backend development? 23

References: 26

1. Introduction

The introduction section describes the purpose of the product specification document.

1.1. Purpose of The Product Design Specification Document

This product design specification document outlines the system design and architecture of SimpleFocals e-commerce web application. This document will be used so that the stakeholders involved in development(project team and project manager), will have a solid understanding of the system architecture. The user interface section of this document will provide the stakeholders who funded the project and insight into current development, leaving open the possibilities of changes depending on the input of the client.

2. General Overview and Design Guidelines/Approach

The general overview section describes the assumptions, constraints and standards that the application must be developed by.

2.1. Assumptions/Constraints/Standards

The functionalities of the system described in the ***Requirements Specification Document (3. Requirements Specification)*** will be implemented. As outlined in that document the user will require a computing device(computer, tablet or smartphone), it is assumed that the user's device has an Internet browser connected to the internet. The web application should be able to handle 100 concurrent users at a given time and the system should be responsive to the requested functions of the user. In the event that the system cannot complete the function available, the system shall display appropriate information to the user as to why the function cannot be completed.

The application will be developed in a mobile first approach to design, ensuring that the user experience and features on each page of the web application remain consistent independent of device. The web application will be developed using Chrome Browser tools where testing for different device screen sizes is possible(*fig2.1.1*). At the completion of the project the web application will be mobile responsive and desktop ready.

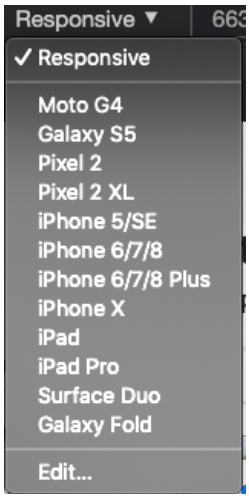

fig2.1.1: options for displaying different screen resolutions on the chrome browser

3. Architecture Design

The architecture design section describes the architectural design of the application. This includes exploring the logical view, the hardware architecture, the software architecture, the security architecture, the communication architecture and the performance of the system.

3.1. Logical View

The logical view outlines the overall architecture of the application. This section will also show the site-view that the user will navigate when interacting with the system.

A simple depiction of SimpleFocals application architecture is outlined in in the image above(*fig3.1.1*). The web application will be developed as a spring boot application. The client in the diagram is a representation of the user's device browser. The user connects to the web application by entering a URL corresponding to the SimpleFocals hosted website. The user can navigate to different webpages in the application. When the user requests a specific resource from the web application, it sends a request to the server, the application's controller will process that request. The controller communicates with the service layer to handle the application logic. The business rules define what the application logic and the functionality should do. The service layer uses repository classes help retrieve/modify the entity's data that is stored in the database and return a result back to the controller and ultimately the view. This communication flow is hidden from the user. The image(*fig3.1.2*) shows the layered view fo the SimpleFocals system architecture.

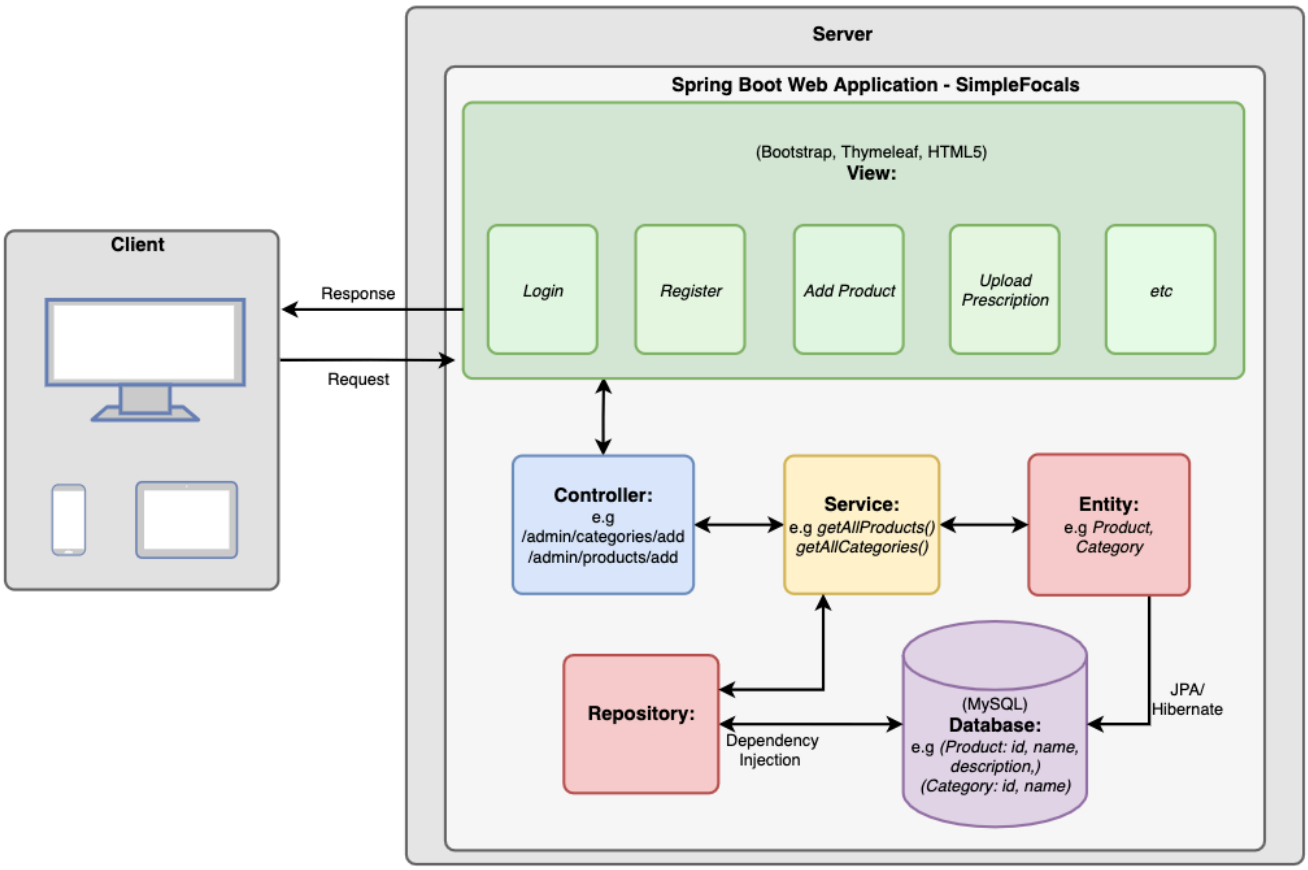

fig3.1.1: System architecture view of SimpleFocals web application

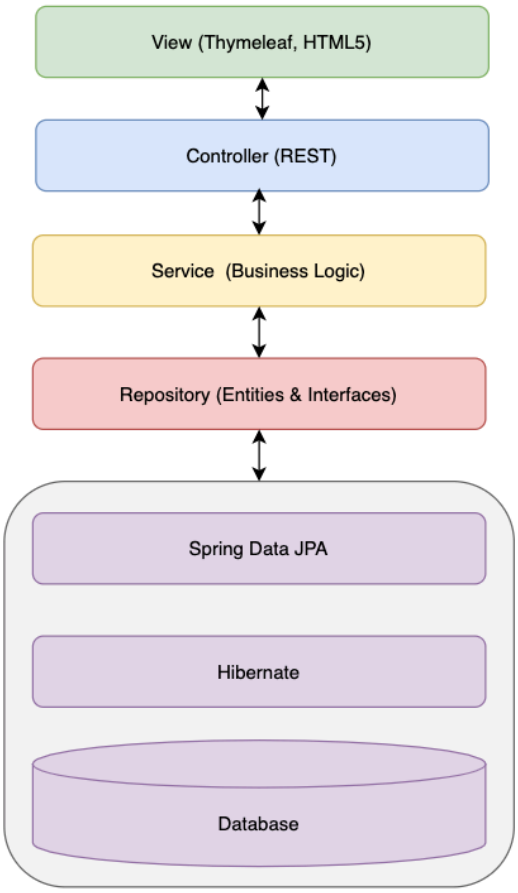

fig3.1.2: Layered System Architecture of SimpleFocals web application

A simple depiction of SimpleFocals site-view for a customer is outlined in in the image above(*fig3.1.3*). The homepage is the entry point of the web application, the customer can access links to the login/register, shop, cart, account and contacts from the navigation bar at the top of all of the SimpleFocals pages. If the customer wishes to access their account, registration and logging into the system is required. When logged into the system the user can access their account page to edit their customer profile, access their cart page, go to checkout and also payment. A simple depiction of SimpleFocals site-view for the administrator is outlined in in the image above(*fig3.1.4*). Once logged into the system, the administrator has access to the content management system pages. A category page is accessible where new categories can be added to SimpleFocals products. A product page is accessible where new products can be added within a specific category.

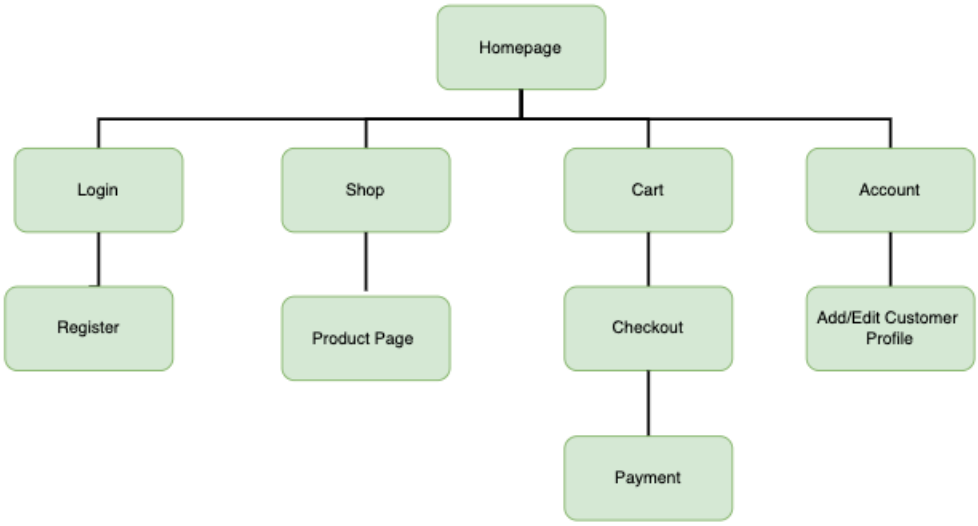

fig3.1.3: Simple site-view for the customer of SimpleFocals

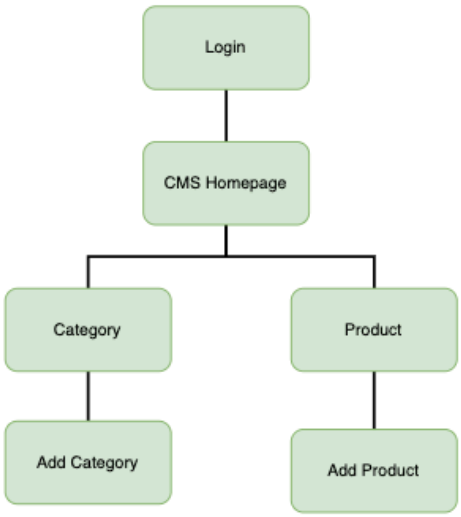

fig3.1.4: Simple site-view for the administrator of SimpleFocals

3.2. Hardware Architecture

The web application will be deployed on Heroku's cloud platform. Since we are in the prototyping stage of application development, Heroku's "*fully-managed platform gives the simplest path to delivering apps quickly.*" (*Heroku, 2021*)

Heroku was chosen as the deployment platform because the free tier offers a simple set-up and configuration that speeds up the process of deployment. Heroku also makes it easy to manage and scale the application in future. For the project Heroku's platform satisfies all the performance and response time requirements that were outlined in ***Requirements Specification Document(3.2.1. Performance/Response time requirement).***

Specification Document.

3.3. Software Architecture

The overall system architecture is displayed in the image(*fig3.1.1*), the technologies that will be used to build this application are listed below.

3.3.1. Front-End

These front-end technologies and resources will be used in the development of the web application.

Bootstrap4.0: Bootstrap is a CSS framework that was used to develop the layout webpages for the application. It has the benefit of being mobile responsive and it aids in quick development of websites in multiple form factors.

HTML5: HyperText Markup Language HTML is the standard language for displaying information on a web browser. Bootstrap classes will be used in combination with HTML tags to define the layout of the e-commerce store.

CSS: Cascading Style Sheet CSS, is used for the presentation of design styles in the web application.

Javascript: Javascript is a language that is used to manipulate objects in the DOM.

JQuery: JQuery is a feature rich Javascript library that is used to manipulate the DOM.

Font Awesome: Online resource for icons that are embedded in the webpages. Font awesome will be used for the facebook, instagram, twitter and linkedin fonts displayed on the website.

3.3.2. Back-End

These back-end technologies and resources will be used in the development of the web application.

Java11: Java is a high level object orientated programming language that was used in the development of the back-end of the application.

Spring boot: Spring boot is a subset of the Spring Framework that allows developers to concentrate on coding applications quickly without having to deal with complex project configurations. Spring boot is the ideal platform for developing for web application in Java. Dependancies can be added to the project as and when specific functionalities are needed.

MySQL DB: MySQL DB is free open-source relational database software that is developed by Oracle. This database was used as the persistent storage in the application.

Hibernate: JPA (Java Persistence API) is the standard approach for Object Relational Mapping(ORM), ORM is where Java objects are mapped to data in a relational database. Hibernate is one of the most common implementations of JPA.

Spring-Boot-Starter: Spring boot starter is a stable of most Spring boot applications, it helps reduce the time the developer uses for searching and adding dependancies. Spring-boot-starter groups the require dependancies together in a single package.

-Web: Spring-Boot-Starter-Web is a library that is used for building RESTful web applications using Spring MVC. Spring MCV applications use Tomcat as the default web container.

-Data-JPA: Spring-Boot-Starter-Data-JPA is a data access abstraction that is used to reduce boiler plate code when implementing data access layer for persistence stores. In the project hibernate was used in combination with JPA libraries for persistent storage in the application's database.

-Security: Spring-Boot-Starter-Security is an authentication and access-control and framework that has become the standard for securing Spring applications. These libraries were used for the login and registration components of the application.

-Thymeleaf: Spring-Boot-Starter-Thymeleaf includes the all the libraries for Thymeleaf for use with Spring applications. Thymeleaf is a HTML5/XHTML/XML server side java templating engine. It was used in the development process to embed objects/data from the backend to front-end views within HTML tags.

-Test: Spring-Boot-Starter-Test is a set of libraries that support test classes. This was used in combination with classes in Spring-Boot-Dev tools to test methods in the development of the application.

Spring-Boot-Devtools: Spring-Boot-DevTools provided development tools for reloading the project and debug options.

Stripe API: Stripe API is the application programming interface provided by trusted third party Stripe in order to provide secure payment processing.

GSON: GSON Google's JSON serialiser/deserialiser was used in place of the native Jackson serialiser/deserialiser package that is default to Spring Boot applications so that the Stripe API wouldn't have any conflicts.

3.4. Security Architecture

The web application will have secure login for users. Users can be logged in using their email and password combination. A users credentials shall be verified by Spring Security. Spring Security shall be used for role based authorisation. Registered customers shall only be able to access a customer's view of the application including all the functionalities that are available to a

registered customer. These functionalities include, access to cart and making a payment for goods. The administrator will have access to the CMS view of the application and have access to all the functionalities of the CMS. This includes adding a product to the SimpleFocals shop. The benefits to using Spring Security is that it can support both authorisation and authentication, it also protects against malicious clients that want to use attacks such as clickjacking, session fixation and cross site request forgery. Spring Security easily integrates into Spring MVC applications.

3.5. Communications Architecture

The basis for SimpleFocals system architecture in the image(*fig3.1.1*), is the model-view-controller(MVC) architecture. Web applications developed using MVC have the benefits of having the presentation, the persistence and control layers loosely coupled and well organised.

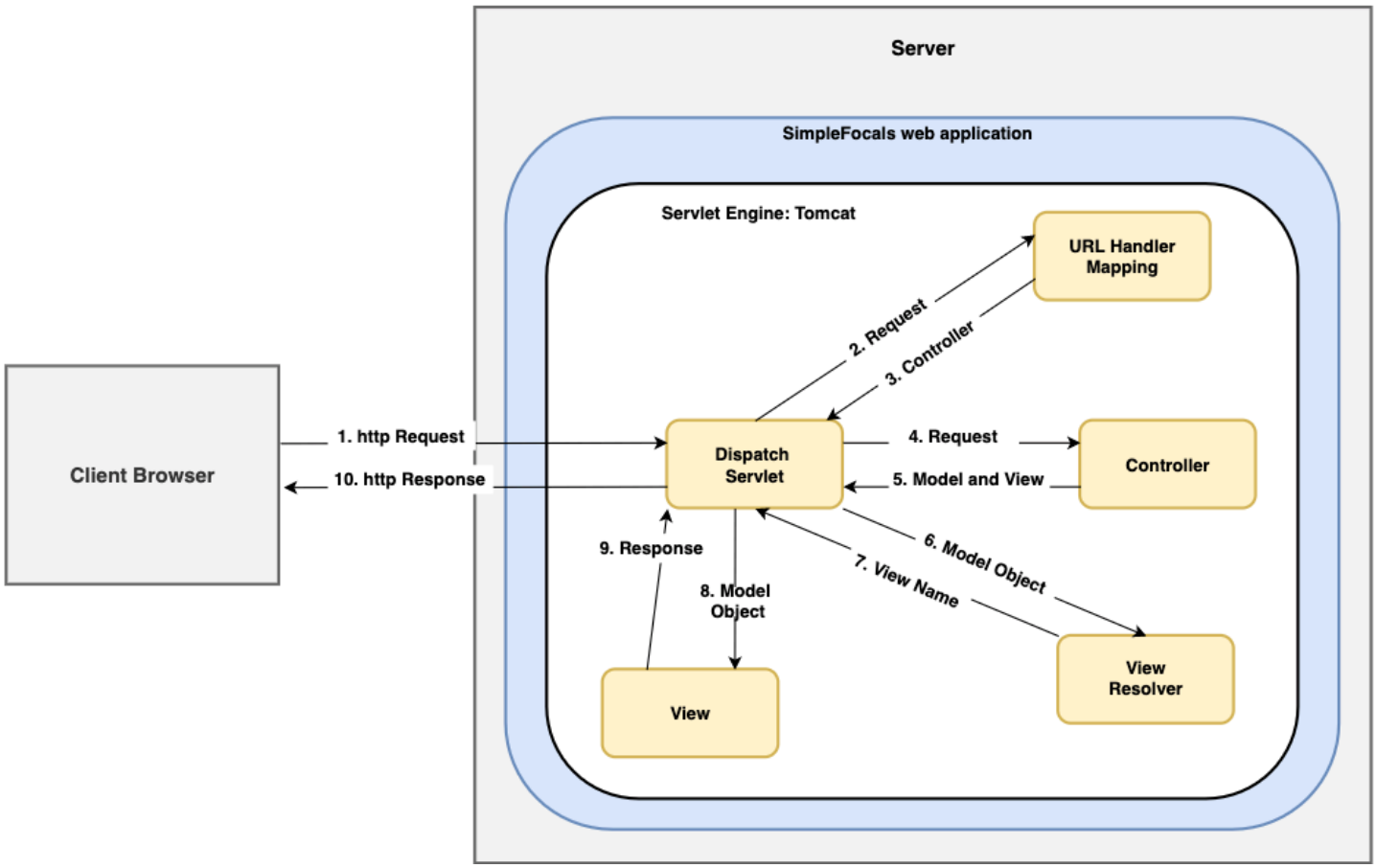

fig3.5.1: How the application handles web requests

When the web application is started on the server, an instance of the application in a servlet container is created to handle the incoming requests and to serve the responses.
In Spring MVC applications, such as SimpleFocals, there is a set flow in the communication structure from the client making a request to them receiving a response. The image(fig3.5.1) outlines the flow of communication in the Spring application.

4. The client sends a http request specifying SimpleFocals web URL in the browser.
5. The request is received by the Dispatcher Servlet. The Dispatcher Servlet acts as a front controller for the application.
6. The Dispatcher Servlet makes use of Spring's URL Mapping Handler to find the specific controller class that the request should be routed for processing. e.g "/admin/products", would route the request to an admin controller class, where it can access a products page.
7. The Dispatcher Servlet forwards the request to the specific controller class the URL Handler Mapping has pointed at. e.g admin controller class
8. The controller class is where implementations of API methods for http requests reside. Those methods include GET and POST methods for http requests. The controller class will call methods in the service layer to handle the logic(*fig3.1.1*). A Model and View object will be passed back to Dispatcher Servlet.
9. The Dispatcher Servlet now sends the Model object to the View Resolver.
10. The View Resolver now returns a view name for the Model object to the Dispatcher Servlet.
11. The Dispatcher Servlet now can route the Model object to the specific view.
12. The Model object containing the data which needs to be displayed in the view can now be rendered. (In SimpleFocals web application we use Thymeleaf templating engine to embed JSON objects/data into HTML5 code).
13. The view can now be returned to the clients browser. Passing the view as http response for rendering in the clients browser completes the communication flow.

3.6. Performance

When interacting with the application the client should be able to access all the functions of SimpleFocals e-commerce website without any noticeable delays in performance. The application when deployed to Heroku cloud will be able to handle the Performance/Response time requirement outlined in ***Requirements Specification Document(3.2.1. Performance/ Response time requirement).***

The web application when deployed should handle 100 users at a given time. The following conditions are expected to be satisfied:

- Login shall not take longer than 3 seconds.
- Loading of pages shall not take longer than 5 seconds

4. System Design

4.1. Use Cases

The use-cases for SimpleFocals web application can be found in the ***Requirements Specification document(3.1. Functional Requirements).***

4.2. Database Design

The database design describes how data saved to the system will be organised and persisted. The conceptual design is the first step in the process in order to develop a relational database system for SimpleFocals. Initially the core entities of the business need to be identified. The relationships and the cardinalities between the entities need to be defined. An entity relationship diagram of the system can then be made describing the overall database system. Lastly we can separate the tables into logical form.

4.2.1. Core entities of the business

In this section we identify the entities of our system. The entities have attributes associated with them. SimpleFocals RDMS stores the following records in tables with the specified primary key identifiers(*fig4.2.1*).

Category		
Attribute	**Entity Type**	**Primary Key**
Category ID	**Strong Entity:** A Product can only be added to an existing Category.	Category ID
Name		

Product		
Attribute	**Entity Type**	**Primary Key**
Product ID	**Weak Entity:** A Product must be part of a Category to exist.	Product ID
Name		
Description		
Price		
Image Name: The table will only store the name of the image being uploaded.		

Role		
Attribute	**Entity Type**	**Primary Key**
Role ID	**Strong Entity:** A Role can exist without a User.	Category ID
Name		

User		
Attribute	**Entity Type**	**Primary Key**
User ID	**Weak Entity:** A User has to have a Role on the system.	User ID
First Name		
Last Name		
Email		
Password		

Customer Profile		
Attribute	**Entity Type**	**Primary Key**
Customer Profile ID	**Strong Entity:** A Customer Profile can be registered on the web application and the Customer Profile doesn't have to ever add Cart Items.	Customer Profile ID
Address		
County		
Postcode		
Email		
Right Eye Prescription		
Left Eye Prescription		
Prescription Image Name: The table will only store the name of the prescription file being uploaded.		

Cart Items		
Attribute	**Entity Type**	**Primary Key**
Cart Items ID	**Weak Entity:** Cart Items is reliant on Customer Profile to add Cart Items.	Cart Items ID
Quantity		

Order History		
Attribute	**Entity Type**	**Primary Key**
Order History ID	**Weak Entity:** Order History is reliant on customer making a purchase.	Order History ID
Date Created		
Quantity		
Subtotal		

fig4.2.1: The Tables of SimpleFocals RDMS

4.2.2. Relationships and cardinalities

The relationships and cardinalities between the tables in SimpleFocals RDMS defines the business rules and assumptions that are to be applied(*fig4.2.2*).

Relationships	Cardinalities
Category contains Product/s: A Category contains one or more Products. A Product is contained within a Category.	**1:M**
User has Role: A User can have one or more Roles. Roles can be assigned to many Users.	**M:N**
User has Customer Profile: A User can have a single Customer Profile. A Customer Profile can only be owned by one User.	**1:1**
Customer Profile adds Cart Items: A Customer Profile adds many Cart Items. Many Cart Items can be added by a Customer Profile.	**1:M**
Cart Items contains Products: Cart Items contains one or many products. One or many Products can be contained in Cart Items.	**1:M**
Customer Profile has Order History: A Customer Profile has one or more Order histories. Many Order Histories can belong to a Customer Profile.	**1:M**
Order History contains Products: An Order History has one or more Products on it. Many products can be on an Order History.	**1:M**

fig4.2.2: The Relationships and Cardinalities of SimpleFocals RDMS

4.2.4. Entity Relationship Diagram.

The Entity Relationship Diagram depicts the system's entities their relationships and cardinalities(*fig4.2.4*).

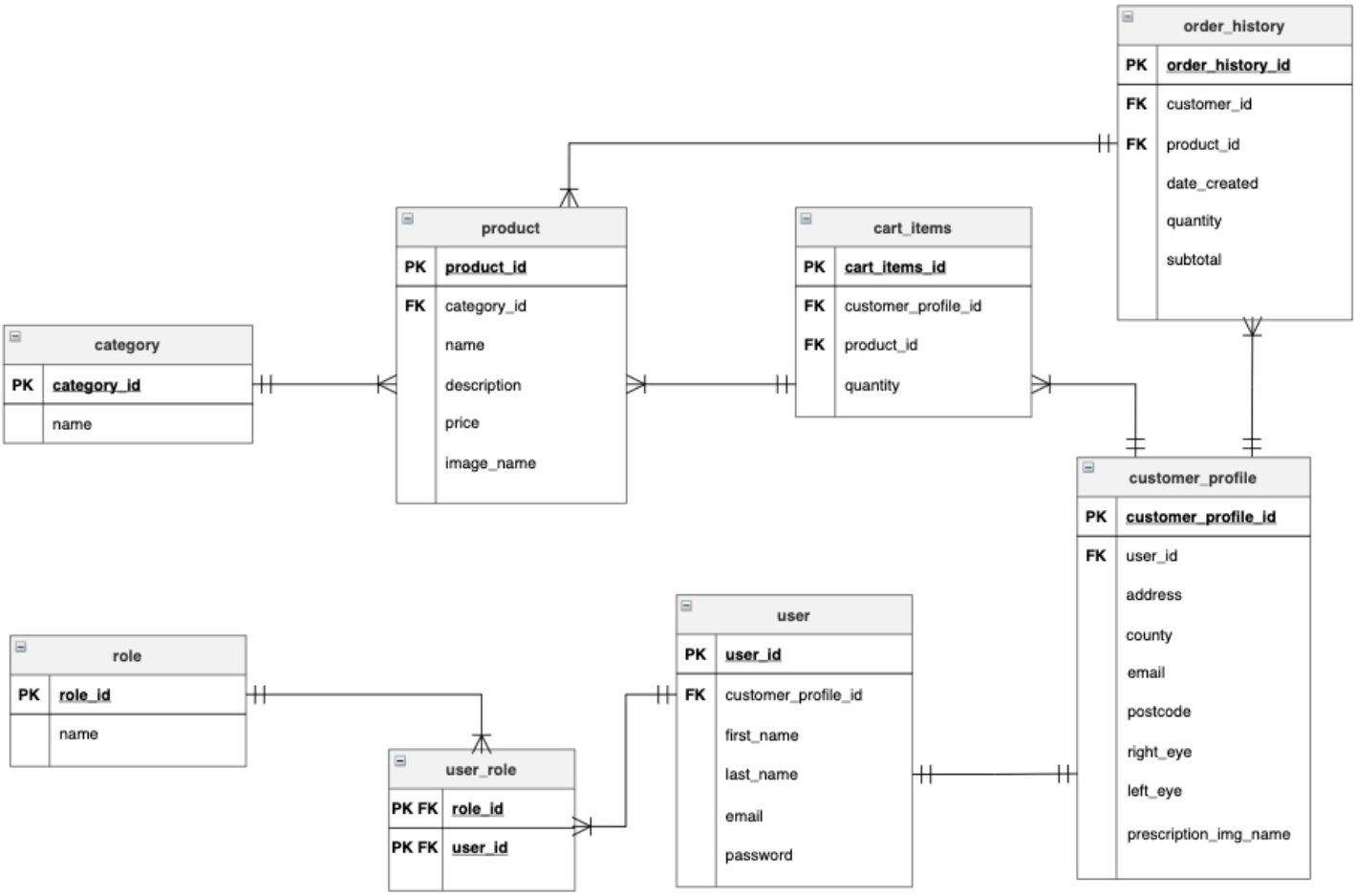

fig4.2.4: The ERD for SimpleFocals RDMS

4.2.5. Logical Tables

This section outlines the normalised tables in the database, the primary, foreign key relationships and the data types associated with the attributes(*fig4.2.5*).

Category	
category_id(PK)	**name**
int	**varchar(255)**

Product					
product_id(PK)	**category_id(FK)**	**name**	**description**	**price**	**image_name**
int	**Int**	**varchar(255)**	**varchar(255)**	**float**	**varchar(255)**

User				
user_id(PK)	first_name	last_name	email	password
int	varchar(255)	varchar(255)	varchar(255)	varchar(255)

Role	
role_id(PK)	name
int	varchar(255)

User_Role	
user_id(PK FK)	role_id(PK FK)
int	Int

Customer Profile				
Customer_profile_id(PK)	user_id(FK)	address	county	Email
int	Int	varchar(255)	varchar(255)	varchar(255)

Customer Profile (continuted)		
left_eye	right_eye	prescription_img_name
double	double	varchar(255)

Cart Items			
cart_items_id(PK)	customer_profile_id(FK)	product_id(FK)	Quantity
int	int	int	varchar(255)

Order History				
order_history(PK)	customer_profile_id(FK)	product_id(FK)	quantity	subtotal
int	int	int	int	double

fig4.2.5: The logical tables of SimpleFocals database system.

4.2.6. Database service and Spring Boot integration.

The Spring boot application is using a MySQL database to store data. MySQL is a relational database that is deployable for cloud applications. The Spring boot application is using Spring

Data JPA and Hibernate for Object relational mapping (ORM) to persist data in the MySQL database. JPA is Java's persistence API that defines a specification for reading, persisting and managing data from objects in Java to relations in a database. Hibernate is an implementation of JPA. Spring Data JPA is Spring's layer on top of Java that makes it easier to implement JPA. Spring Data JPA allows a DAO interface to be defined by extended its repositories using Jpa-Repository this provides a DAO implementation during runtime.

4.2.7.Class/Relation Implementation.

This section shall introduce the implementation the Category class using Spring Data JPA and Hibernate. In the image below(*fig4.2.7.1*), a Category class in Java is defined. The "@Entity" annotation outlines that this class will be a relation in the database that will be automatically generated by Spring Data JPA and Hibernate in the system. The primary key will be unique and incrementally generated by specifying the "@GeneratedValue" annotation and strategy type of "GenerationType.AUTO". There are two columns in this table, the primary key "category_id" of type integer and "name" that is of type String. Constructors, getters and setters are defined in order to access the object and member variables in service layer of the application.

```
@Entity
@Table(name = "category")
public class Category {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    @Column(name = "category_id")
    private int id;

    private String name;

    public Category() {
    }

    public Category(int id, String name) {
        this.id = id;
        this.name = name;
    }
```

fig4.2.7.1: Category class

The table generated from the Category class is shown in the image(*fig4.2.7.2*) with a Men's Glasses and a Women's Glasses category.

	category_id	name
1	2	Mens Glasses
2	4	Womens Glasses

fig4.2.7.2: The Category table that maps to Category class

In the image below(*fig4.2.7.3*), a CategoryRepository Interface is defined that extends JpaRepository. This allows the Category class the use of methods that implement CRUD functionality on the database tables when the business logic is defined in the service layer.

```
import org.springframework.data.jpa.repository.JpaRepository;

public interface CategoryRepository extends JpaRepository<Category, Integer> {
}
```

fig4.2.7.3: CategoryRepository Interface

In the image below(*fig4.2.7.4*), a CategoryService class is defined. Here the defined CategoryRepository object has many methods to facilitate CRUD functionality. The Category-Repository methods will be used in the service layer to handle the application logic.

```java
//the Service is where the business logic should go

@Service
public class CategoryService {
    @Autowired
    CategoryRepository categoryRepository;

    public List<Category> getAllCategory(){
        return categoryRepository.findAll();
    }

    public void addCategory(Category category){
        categoryRepository.save(category);
    }

    public void removeCategoryById(int id){
        categoryRepository.deleteById(id);
    }

    public Optional<Category> getCategoryById(int id){
        return categoryRepository.findById(id);
    }
}
```

fig4.2.4: CategoryService Class

4.3. Data Conversions

4.3.1. Back-End

Using Spring Data JPA and Hibernate, data class objects in Java are mapped to data relational database tables in MySQL. With such mappings the system can perform CRUD operations on data in the database without writing any SQL.

4.3.2 Front-End

When some dynamic function is requested from the front-end, the client sends the server JSON objects(*fig4.3.1*). Once sent to the server the application parses the information into java objects where it can be processed and stored in the database.

```json
{
    "category_id": 5,
    "name": "Womens Glasses"
}
```

fig4.3.1: JSON object

4.4. Application Programming Interfaces

The application will be integrating Stripe API for payment processing. Customers will be able to pay for SimpleFocals products by credit or debit card. Stripe ensures all payments on orders shall be securely processed on their servers without our web application storing any of the customers sensitive card information. The steps for implementation will require the application to:

1. Add Stripe dependancy to the project.
2. Build the checkout page loading the Stripe.js script. This script is used to access the API keys on the client and the card input field on the form.
3. Complete the payment on the client. The Stripe handles the API response.
4. Test the API using test card numbers supplied by Stripe.

4.5. User Interface Design

The user interface will be developed using frontend technologies HTML, CSS and Javascript. In order to speed up development time, the project shall use Bootstrap to design the pages of the website. Bootstrap is a CSS grid based framework that allows rapid development using Bootstrap's predefined classes that allow for mobile responsive development of websites. The dynamic elements of the user interface will use thymeleaf's templating engine to update information on the webpages. Examples of the wireframes developed for this project are detailed in Interface requirement in the ***Requirements Specification Document(4.1. Graphical User Interface (GUI)).***

4.6. Performance

There are a number of reasons why Java is a good choice for backend development.

4.6.1. Why use java for backend development?

- Java is an object orientated high level language that uses a virtual machine to compile programs, this allows it the flexibly to run on many operating systems and platforms.
- Java is scalable and robust, it has the benefits of automatic memory management and garbage collection which can speed up web application development.
- Java has lot of support for open source libraries such XML and JSON parsing libraries.
- Java supports frameworks and tools for quickly developing back-end applications. Spring, Apache and Hibernate being examples of some.

References:

Heroku, 2021. *Developers, teams, and businesses of all sizes use Heroku to deploy, manage, and scale apps [online]*

Available at: https://www.heroku.com/ [Accessed 1 July 2021]

Baeldung, 2021. *A Comparison Between Spring and Spring Boot. [online]*

Available at: https://www.baeldung.com/spring-vs-spring-boot [Accessed 1 July 2021]

Javapoint, 2021. *Spring Boot Starters. [online]*

Available at: https://www.javatpoint.com/spring-boot-starters [Accessed 8 July 2021]

6.4. Project plan.

This was the project plan I had during the project lifecycle. I followed the plan meeting all the deadlines except the final weeks. The project deadline was extended into the 3rd week of August.

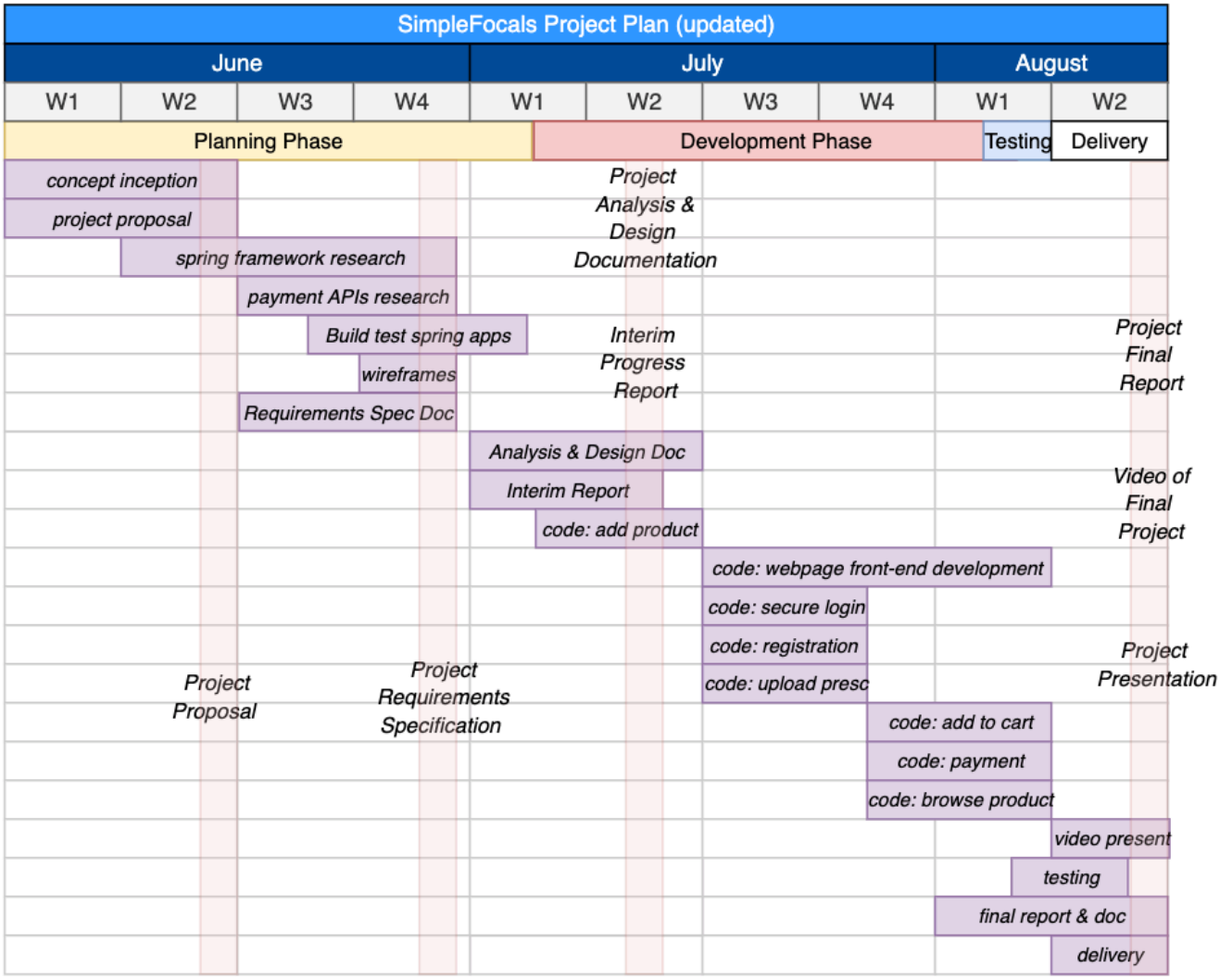

fig4.1: SimpleFocals updated project plan

6.5 Acronyms, Abbreviations and Terms:

ORM: Object–relational mapping in computer science is a programming technique for converting data between incompatible type systems using object-oriented programming languages. This creates, in effect, a "virtual object database" that can be used from within the programming language.

JPA: The Java Persistence API (JPA) is a Java specification for accessing, persisting, and managing data between Java objects / classes and a relational database

API: Application Programming Interface, which is a software intermediary that allows two applications to talk to each other.

CMS: A Content Management System is a computer software used to manage the creation and modification of digital content.

MVC: Model–view–controller is a software design pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements.

URL: Uniform Resource Locator is a reference to a web resource that specifies its location on a computer network and a mechanism for retrieving it.

GET: GET method requests a representation of the specified resource. Requests using GET should only be used to request data.

POST: A POST request is typically sent via an HTML form and results in a change on the server.

DAO: DAO is an abbreviation for Data Access Object, so it should encapsulate the logic for retrieving, saving and updating data in your data storage (a database, a file-system)

CRUD: Create, Read, Update, and Delete are the four basic operations of persistent storage.

JSON: JSON is an open standard file format and data interchange format that uses human-readable text to store and transmit data objects consisting of attribute–value pairs and arrays.

XML: Extensible Markup Language is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable.

CSS: Cascading Style Sheets is a style sheet language used for describing the presentation of a document written in a markup language such as HTML.

HTML: The HyperText Markup Language, or HTML is the standard markup language for documents designed to be displayed in a web browser.

MacOS: Linux based 3rd party Operating System that is developed by Apple and native to Mac computers.

DOM: Document Object Model is an interface that has a trees structure representation of HMTL and XML, where each node is part of the document representation.

REST: Representational State Transfer, is a set of architectural principles that are used for creating web services.

6.5. Other Materials Used:

2.7. Customer Testing - User Acceptance Test

Created By:		Reviewed By:		Version	
Testers' Name:		Date Tested:		Test:(pass/Fail/ Not executed)	
Test Senario:					
Step:	Step Details	Expected Results		Actual Result	Pass/Fail