

# DELUXE VISION

## Table of Contents

### Part 1: Requirements

- 1.1 Introduction
- 1.2 Major Design Decisions
- 1.3 Diagrams and Architecture
- 1.4 Activities/Backlog
- 1.5 Test Driven Development
- 1.6 Design Patterns, Plan, Test Coverage, Description of Server Side
- 1.7 Demo

### 1.1 Introduction

#### Purpose:

The purpose/objective of this project is to design and create an ecommerce website that allows the purchase of glasses.

#### Overview:

Some problems that this website and project solves include:

- Browse and purchase Glasses
- Filtering items by different categories
- Checking out items
- Registering an account to create purchases
- Being able to run reports on sales and application usage as an admin
- Able to try on the glasses without purchasing them

#### References:

- <https://www.zeelool.com/>

## 1.2 Major Design Decisions

a) For our design we decided to use the Model-View-Controller design pattern, otherwise known as the MVC pattern. This design pattern can commonly be seen being used for all sorts of different web development and involves dividing the system into three connected elements. These components are the model, the view and the controller. See Figure 1.

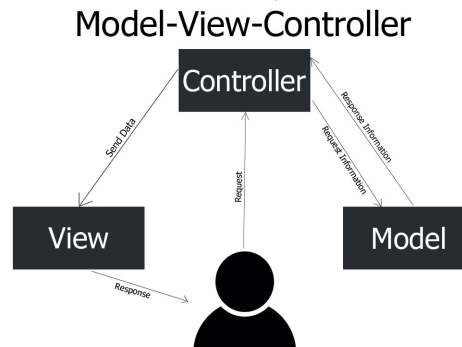
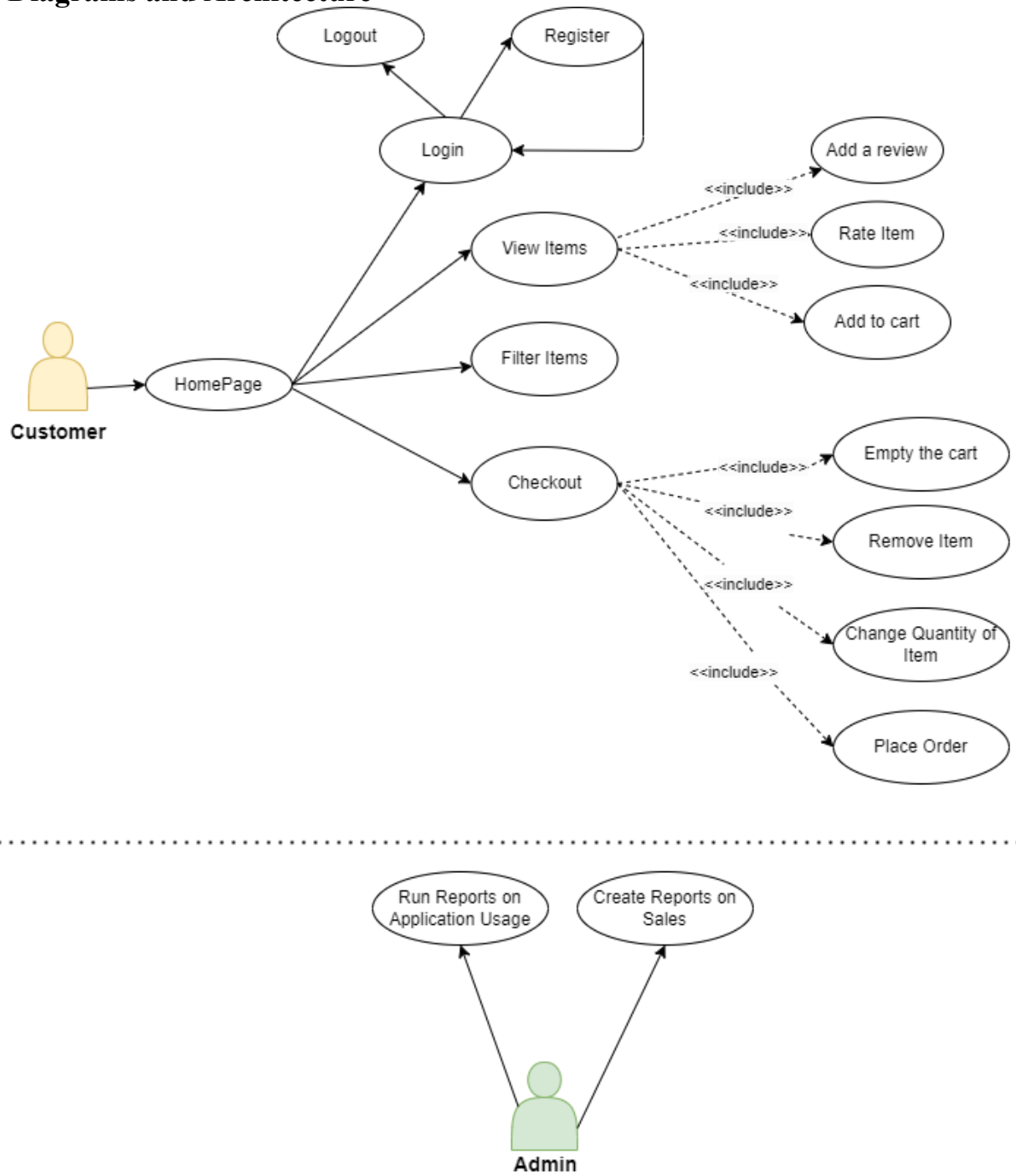


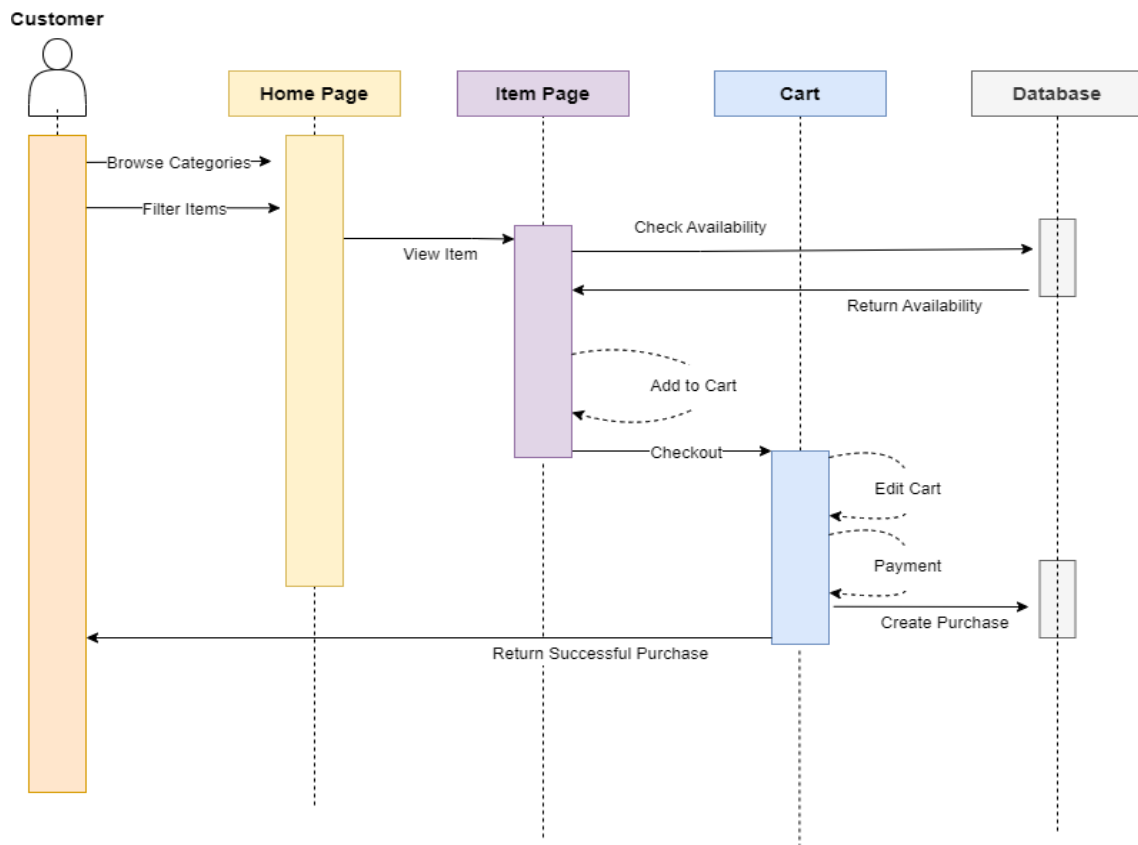
Figure 1. MVC Pattern

The entire system for DeluxeVision is created and designed based on the MVC pattern. The reason we decided to go with this pattern is due to the fact that it is very easy to manage web applications by breaking it up into three parts, it is easy to maintain and easily modifiable, it allows for multiple views and a faster process for development. The view which is created by Angular and HTML includes all of the content that we want to display to the user. The controller helps get the input from the user from the view and grabs data for the model. The model helps to navigate and interact with the database to provide data to the Model to display to users and also updates the database when necessary. These three components allow us to maintain them much easier, it's the separation of concerns that grant us freedom in making sure all parts are running smoothly. Since there is high cohesion between all three components, it allows for things to be used over and over again and since we have separated unrelated parts of the code into different places, we have low coupling.

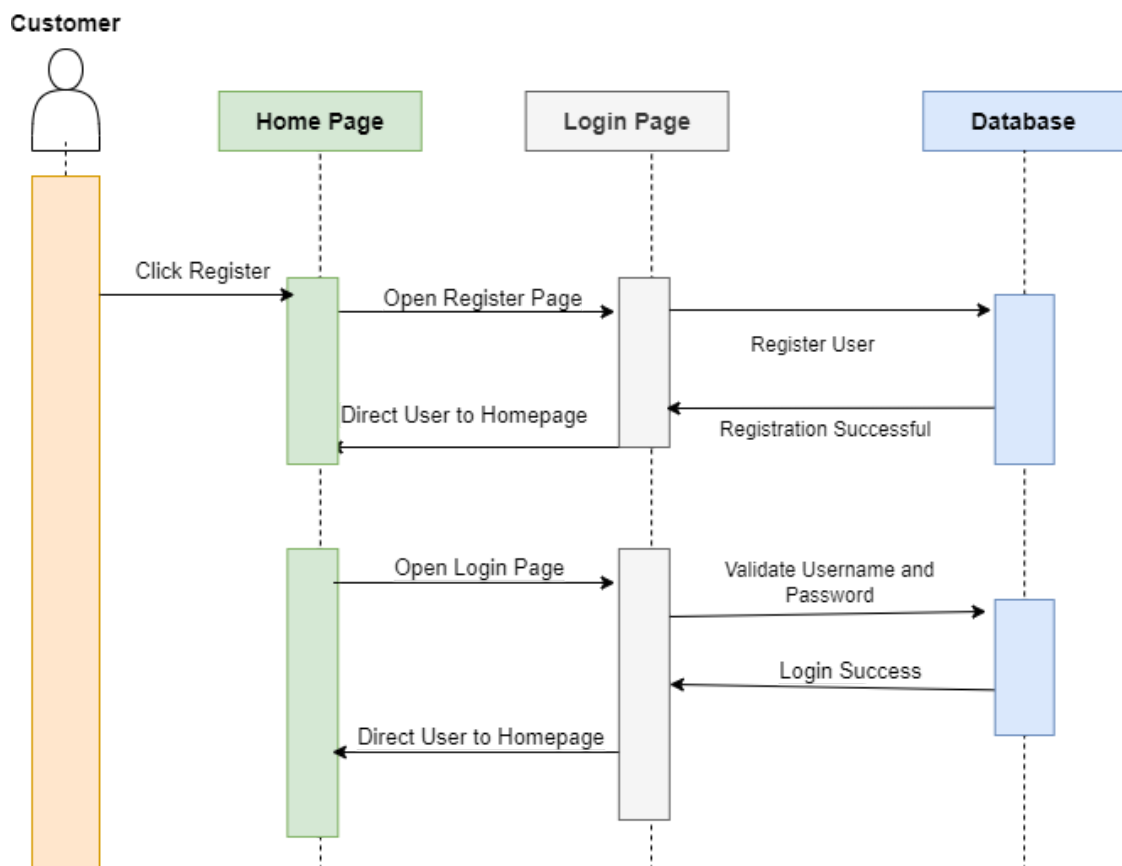
### 1.3 Diagrams and Architecture



**Figure 2:** Use Case Diagram



**Figure 3: Sequence Diagram 1 (Browse/Purchase)**



**Figure 4: Sequence Diagram 2 (Login)**

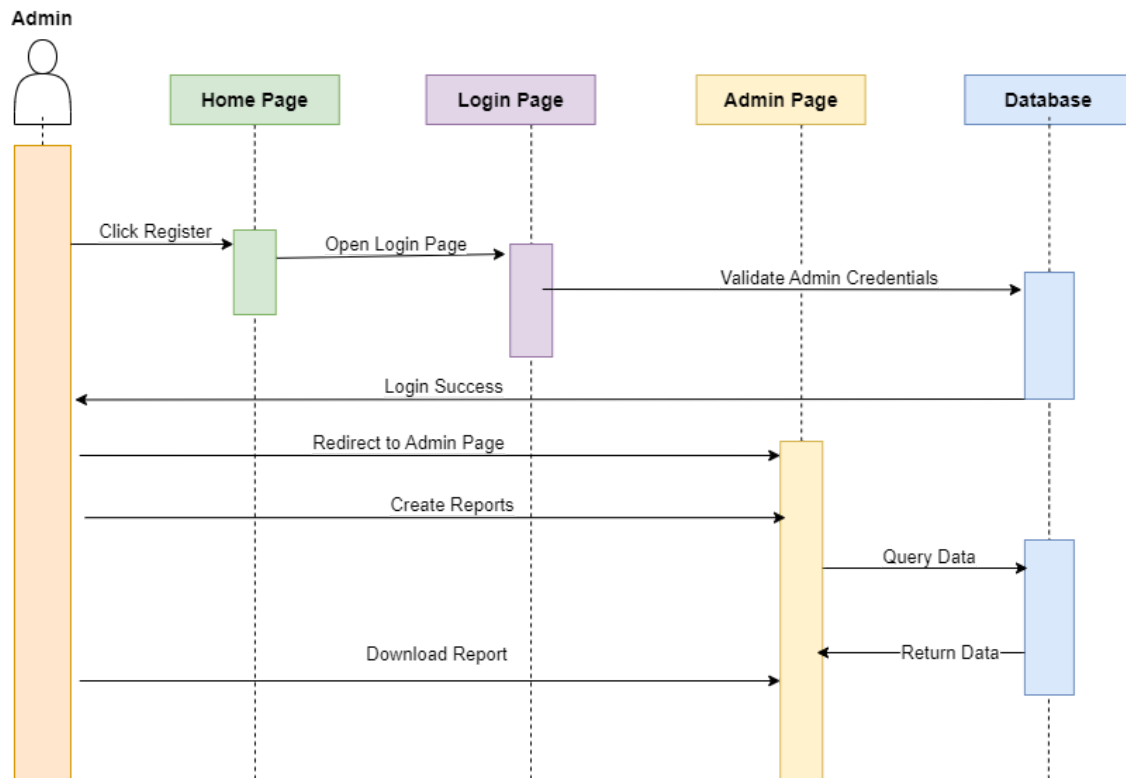
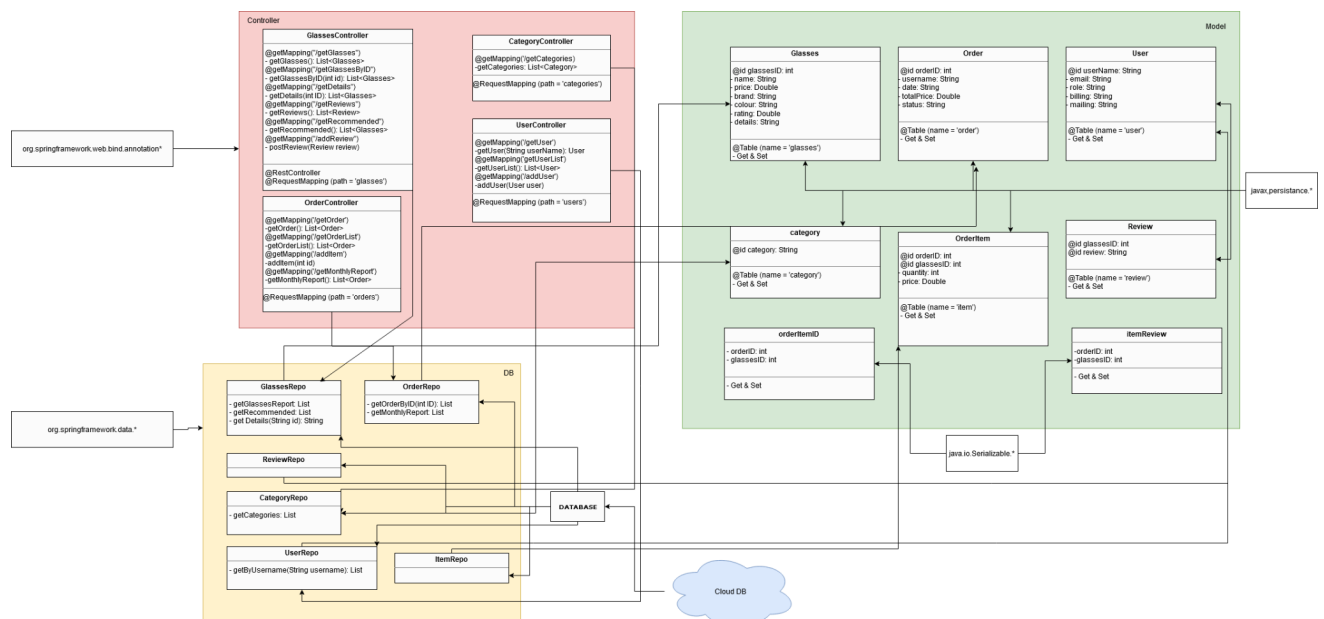


Figure 5: Sequence Diagram 3 (Admin)



<https://drive.google.com/file/d/1Zqr5PPIbJdd2hy-yyUfSYyIDS6gAjZda/view?usp=sharing>

Figure 6: Refined Component Diagram

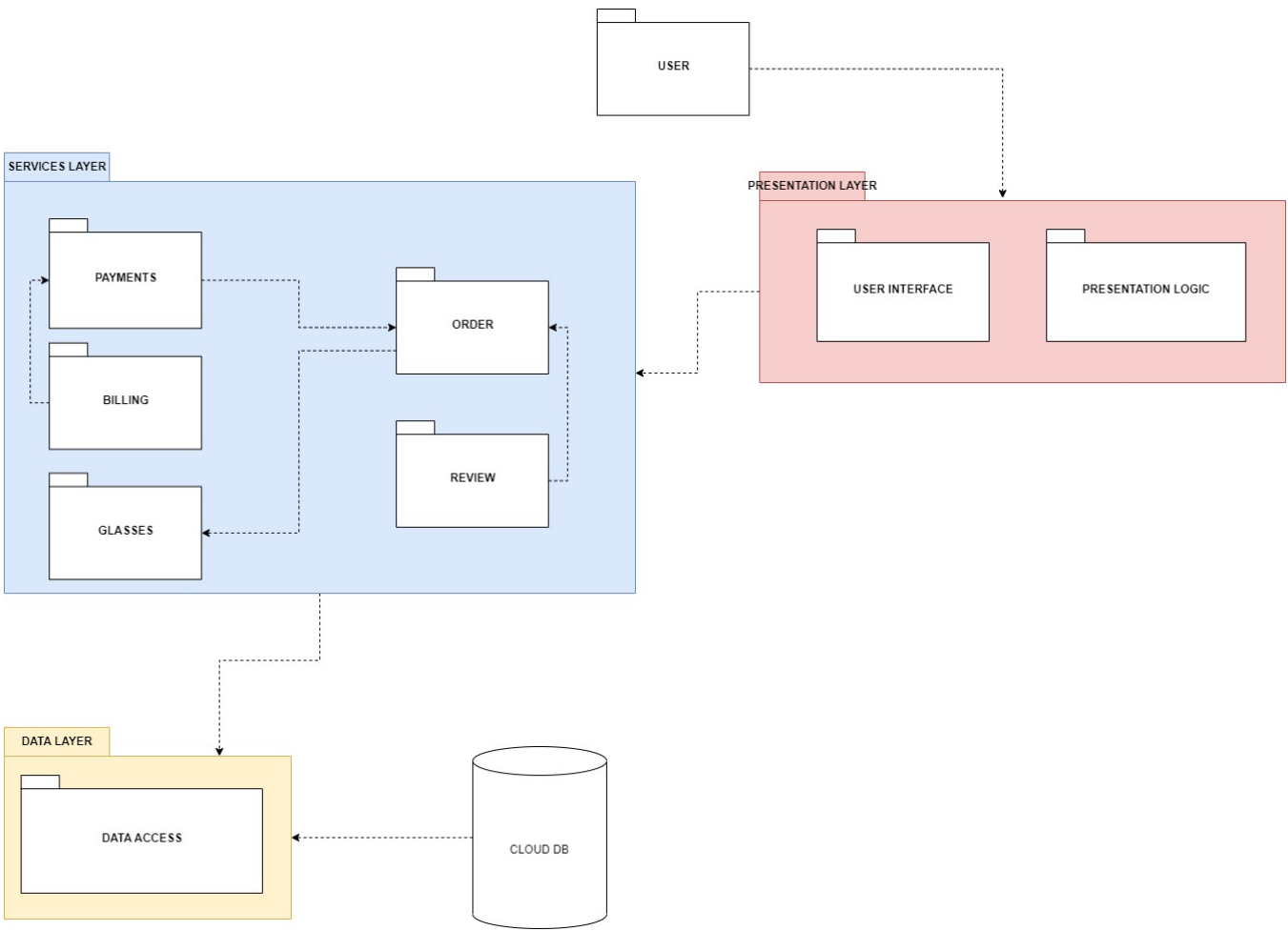


Figure 7: Package Diagram

Modules			
Module Name	Description	Exposed Interface Names	Interface Description
Glasses	The module for all glasses information	Glasses: GlassesController  Glasses:OrderController	Glasses: GlassesController: Grab all info related to glasses/each glasses  Glasses:OrderController: Put glasses data into order
Order	The module for all order information	Order:OrderController  Order:GlassesController	Order:OrderController: Grab all info related to orders  Order:GlassesController: For use when getting glasses info from an order
User	The module for all user information	User:UserController User:OrderController	User:UserController : Grab all data related to User info etc.  User:OrderController: Grab all data regarding a users orders

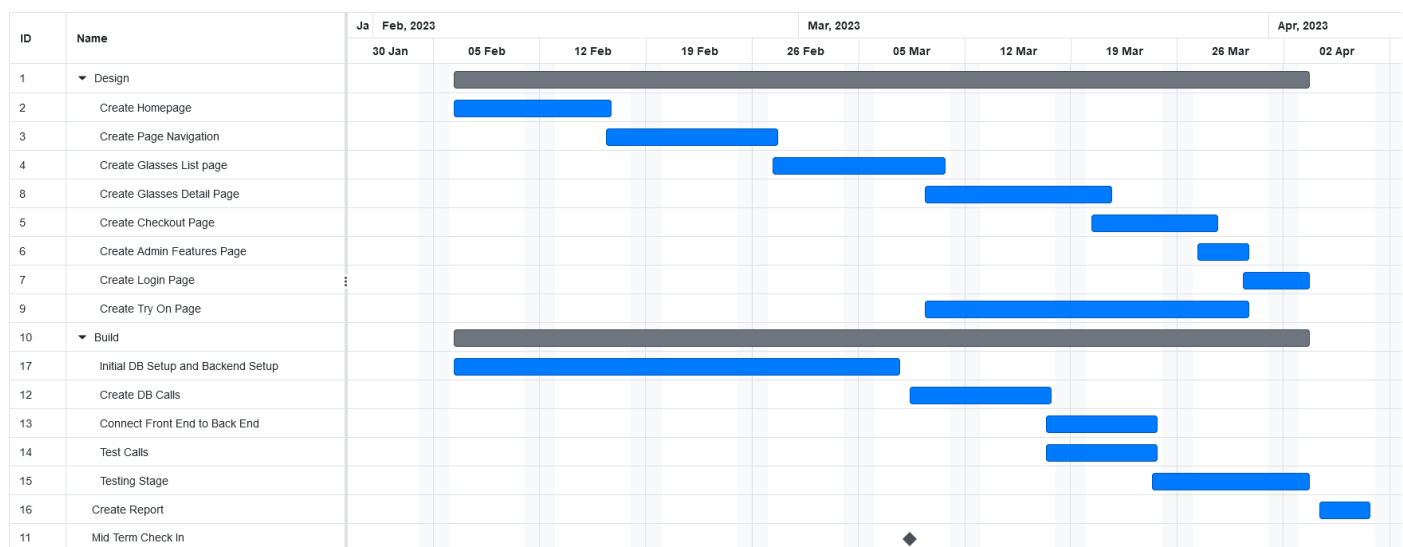
Interfaces		
Interface Name	Operations	Operation Description
Glasses:GlassesController	List<Glasses> getGlasses(): getGlassesByID(int id) getDetails(int id), getRecommended()  used by M1 and M2  List<Review> getReviews() used by M1  postReview(Review review)  used by M1	getGlasses(): Gets a list of all glasses  getGlassesByID(): Gets a list of all glasses with same ID  getDetails(): Gets details of glasses  getRecommended(): Get list of recommended glasses  getReviews(): Gets reviews for glasses  postReview(): Posts a review
Order:OrderController	List<Order> getOrder(), getOrderList(), getMonthlyReport()  used by M2 and M3  addItem(int id)  used by M2	getOrder(): Gets order for checkout etc.  getOrderList(): Gets a full list of all orders for a user  getMonthlyReport(): Admin feature to get all reports for the month  addItem(): Add glasses to order
User:UserController	List<User> getUserList()  User getUser(String username)  addUser(User user)  used by M3	getUserList(): Gets a list of all users  getUser(): get a specific user  addUser(): adds a user to DB, for registration purposes



## 1.4 Activities/Backlog

### Backlog List

- Create Homepage
- Create Page Navigation
- Create Glasses List page
- Create Glasses Details page
- Create Checkout page
- Create Admin Features
- Create Login Page
- Create Try On Page
- Setup DB and Backend
- Create DB calls to frontend
- Connect Front end to back end
- Test Calls
- Test Implementations
- Create Report



**Figure 8: GANTT Diagram**

## 1.5 Test Driven Development

<b>Test ID</b>	<b>0001</b>
<b>Category</b>	<b>Evaluation of Homepage Routing</b>
<b>Requirements Coverage</b>	<b>UI-Successful-Routing</b>
<b>Initial Condition</b>	The webpage has been rendered and runs
<b>Procedure</b>	<ol style="list-style-type: none"><li>1. The user has successfully logged in</li><li>2. The user is presented with the homepage</li><li>3. The user can navigate to any place in the website via the homepage</li></ol>
<b>Expected Outcome</b>	The login form is closed and the user is presented with the main homepage where all buttons lead to someplace/ a different page

<b>Test ID</b>	<b>0002</b>
<b>Category</b>	<b>Evaluation of User Authentication</b>
<b>Requirements Coverage</b>	<b>UI-Successful-Auth</b>
<b>Initial Condition</b>	The webpage has been rendered and runs
<b>Procedure</b>	<ol style="list-style-type: none"><li>1. The user selects the login button</li><li>2. The user provides a username</li><li>3. The user provides a password</li><li>4. If Admin, lead to admin page, if User lead to homepage</li></ol>
<b>Expected Outcome</b>	The login form is closed and the user is presented with the main homepage or the admin page

<b>Test ID</b>	<b>0003</b>
<b>Category</b>	<b>Evaluation of Correct Glasses Details</b>
<b>Requirements Coverage</b>	<b>UI-Item-Detail-Test</b>
<b>Initial Condition</b>	The webpage has been rendered and runs and the user has clicked on a pair of glasses
<b>Procedure</b>	<ol style="list-style-type: none"><li>1. Open the webpage</li></ol>

	<ol style="list-style-type: none"> <li>2. Select a pair of glasses</li> <li>3. The page renders and shows the glasses with the details of the glasses</li> </ol>
<b>Expected Outcome</b>	The webpage should display the glasses selected and the details of the glasses that the user wants to see

<b>Test ID</b>	<b>0004</b>
<b>Category</b>	<b>Evaluation of Reports on File or DB</b>
<b>Requirements Coverage</b>	<b>UI-Report-Successful</b>
<b>Initial Condition</b>	The webpage has been rendered and runs and the user is an admin logged in
<b>Procedure</b>	<ol style="list-style-type: none"> <li>1. The user has logged in to an admin account</li> <li>2. The page renders to the admin page</li> <li>3. The admin clicks the run report button to display report of sales or application usage</li> </ol>
<b>Expected Outcome</b>	Once the admin selects the report they wish to see, the report is rendered to the user and displays a report of sales or application usage

<b>Test ID</b>	<b>0005</b>
<b>Category</b>	<b>Evaluation of User Registration</b>
<b>Requirements Coverage</b>	<b>UI-Successful-Register</b>
<b>Initial Condition</b>	The webpage has been rendered and runs
<b>Procedure</b>	<ol style="list-style-type: none"> <li>5. The user selects the register button</li> <li>6. The user creates a new username</li> <li>7. The user creates a new password with the given criteria</li> <li>8. If the username doesn't exist in DB, it is created with the entered password</li> <li>9. A registration success message is displayed</li> <li>10. The user directed to the home page</li> </ol>
<b>Expected Outcome</b>	The new user is successfully saved in DBA and registration success message is displayed.

<b>Test ID</b>	<b>0006</b>
<b>Category</b>	<b>Evaluation of Product Filtering</b>
<b>Requirements Coverage</b>	<b>UI-Product-Filter</b>
<b>Initial Condition</b>	The webpage has been rendered and runs
<b>Procedure</b>	<ol style="list-style-type: none"> <li>4. Open the webpage on main page</li> <li>5. Navigate to All Products page</li> <li>6. Apply price, color, and shape filters one by one</li> <li>7. The page renders and display the glasses with the selected criteria</li> </ol>
<b>Expected Outcome</b>	The user is able to see the filtered list of glasses

## 1.6 Design Patterns, Plan, Test Coverage, Description of Server Side

\*Refer to figure 6 for refined component diagram

- a) Continuing off of what was written in Section 1.2 we used MVC and DAO Pattern. Another design pattern we decided to use is the Data Access Object (DAO) pattern. As our application involves interaction with the database for data modification and retrieval, DAO provides us with an abstraction layer separating the database access code from our Application's business logic. This allowed us to easily be able to modify and update our functionalities. The main concept of this design pattern is the separation of concerns, but moreover, using DAO pattern improved the flexibility, scalability, and maintainability of our application; causing a more simplified way of testing and providing security. DAO promotes modularization and reduces the coupling we have between different components. Since we encapsulated our database access logic, we can isolate the code that interacts with the database, and this makes it easier to modify, test and maintain the application in its entirety. What we have also done is, the services in angular material will help us connect our backend to our front end to grab all the necessary things for displaying things on our website.

We will also be using AWS to deploy our system, as it is cloud native.

b) Activities Plan: Refer to table in Section 1.5 for meeting logs

The activities done in this part of the sprint were heavily in regards to setting up the backend server side implementation:

- Create Homepage
- Create Page Navigation
- Create Glasses Details page
- Create Checkout page
- Create Login Page
- Setup DB and Backend
- Test Calls
- Test Implementations
- Create Report

c) Test coverage for the server side included many simple tests to make sure that the data that we are trying to collect is the correct data. Essentially what we wanted to make sure was working was that, if we request something that is not in the database, for example, glasses with an id that does not exist, we should get an error. Also we wanted to see that if we made a request to the database, without the proper JSON data to grab the data, it should send an error back to us. Lastly we wanted to see if we did give it the correct data, we should get the correct response back from the backend. Refer to figure 9 for minor implementations of that.

```
▼ GET http://localhost:8080/glasses/all
GET /glasses/all HTTP/1.1
User-Agent: PostmanRuntime/7.31.3
Accept: */*
Postman-Token: 9190e69f-b17f-4863-8160-3ce6391d4b89
Host: localhost:8080
Accept-Encoding: gzip, deflate, br
Connection: keep-alive

HTTP/1.1 200 OK
Content-Type: application/json
Transfer-Encoding: chunked
Date: Fri, 31 Mar 2023 21:07:01 GMT
Keep-Alive: timeout=60
Connection: keep-alive

[{"name":"Ray-Ban RX5356","shape":"Square","imageUrl":"https://www.kits.ca/cart/images/frames/items/5/50/50000001564_IMG.jpg","glassesCode":"61d7c91f-dacc-493d-967d-64a1e4cda862"}]
```

Figure 9. Test 1 for Retrieving All Glasses in DB (Postman)

```
C:\Windows\System32>http GET :8080/glasses/all
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Fri, 31 Mar 2023 21:33:27 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
  {
    "glassesCode": "61d7c91f-dacc-493d-967d-64a1e4cda862",
    "imageUrl": "https://www.kits.ca/cart/images/frames/items/5/50/5000001564_IMG.jpg",
    "name": "Ray-Ban RX5356",
    "shape": "Square"
  },
  {
    "glassesCode": "4c4b3b64-89fd-453b-8d70-558ebc882517",
    "imageUrl": "https://www.kits.ca/cart/images/frames/items/6/68/6800004018_IMG.jpg",
    "name": "Kenneth Cole New York KC0249-3",
    "shape": "Round"
  },
  {
    "glassesCode": "7ac14e42-4e0f-48a7-aa1c-8509f700820d",
    "imageUrl": "https://www.kits.ca/cart/images/frames/items/6/61/6100001165_IMG.jpg",
    "name": "KITS Thursday",
    "shape": "Rectangle"
  },
  {
    "glassesCode": null,
    "imageUrl": "https://www.kits.ca/cart/images/frames/items/6/61/6100001165_IMG.jpg",
    "name": "KITS Friday",
    "shape": "Rectangle"
  },
  {
    "glassesCode": "7ac14e42-4e0f-48a7-aa1c-8509f700820d",
    "imageUrl": "https://www.kits.ca/cart/images/frames/items/6/61/6100001165_IMG.jpg",
    "name": "KITS Friday",
    "shape": "Rectangle"
  }
}
```

Figure 10. Test 2 for Retrieving All Glasses in DB (Httpie)

```
▼ POST http://localhost:8080/glasses/add
POST /glasses/add HTTP/1.1
Content-Type: application/json
User-Agent: PostmanRuntime/7.31.3
Accept: */*
Postman-Token: 2b078984-cdec-426f-8b15-469b44fac9ab
Host: localhost:8080
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 160

{"name":"Kenneth Cole New York KC0249-3",
"shape":"Round",
"imageUrl":"https://www.kits.ca/cart/images/frames/items/6/68/6800004018_IMG.jpg"}

HTTP/1.1 201 Created
Content-Type: application/json
Transfer-Encoding: chunked
Date: Fri, 31 Mar 2023 21:15:46 GMT
Keep-Alive: timeout=60
Connection: keep-alive

{"name":"Kenneth Cole New York KC0249-3","shape":"Round","imageUrl":"https://www.kits.ca/cart/images/frames/items/6/68/6800004018_IMG.jpg","glassesCode":"4c4b3b64-89fd-453b-8d70-558ebc882517"}
```

Figure 11. Test 3 for Adding Glasses Data into the DB directly in response body (Postman)

```
▼ POST http://localhost:8080/glasses/add
POST /glasses/add HTTP/1.1
Content-Type: application/json
User-Agent: PostmanRuntime/7.31.3
Accept: */*
Postman-Token: e912688f-9878-4bd2-819a-efb524b20920
Host: localhost:8080
Accept-Encoding: gzip, deflate, br
Connection: keep-alive
Content-Length: 145

{"name":"Ray-Ban RX5356",
"shape":"Square",
"imageUrl":"https://www.kits.ca/cart/images/frames/items/5/50/5000001564_IMG.jpg"}

HTTP/1.1 201 Created
Content-Type: application/json
Transfer-Encoding: chunked
Date: Fri, 31 Mar 2023 21:06:03 GMT
Keep-Alive: timeout=60
Connection: keep-alive

{"name":"Ray-Ban RX5356","shape":"Square","imageUrl":"https://www.kits.ca/cart/images/frames/items/5/50/5000001564_IMG.jpg","glassesCode":"61d7c91f-dacc-493d-967d-64a1e4cda862"}
```

Figure 12. Test 4 for Adding Glasses Data into the DB directly in response body (Postman)

```

C:\Windows\System32>http POST :8080/glasses/add < data.json
HTTP/1.1 201
Connection: keep-alive
Content-Type: application/json
Date: Fri, 31 Mar 2023 21:29:03 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
  "glassesCode": "7ac14e42-4e0f-48a7-aa1c-8509f700820d",
  "imageUrl": "https://www.kits.ca/cart/images/frames/items/6/61/61000001165_IMG.jpg",
  "name": "KITS Thursday",
  "shape": "Rectangle"
}

```

Figure 13. Test 5 for Adding Glasses Data into the DB from a file (Httpie)

```

C:\Windows\System32>http PUT :8080/glasses/update < data.json
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Fri, 31 Mar 2023 21:30:36 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
  "glassesCode": "7ac14e42-4e0f-48a7-aa1c-8509f700820d",
  "imageUrl": "https://www.kits.ca/cart/images/frames/items/6/61/61000001165_IMG.jpg",
  "name": "KITS Friday",
  "shape": "Rectangle"
}

```

Figure 14. Test 6 for Updating a Glasses Data in the DB using a Json (Httpie)

```

C:\Windows\System32>http DELETE :8080/glasses/delete/4
HTTP/1.1 500
Connection: close
Content-Type: application/json
Date: Fri, 31 Mar 2023 21:32:54 GMT
Transfer-Encoding: chunked

{
  "error": "Internal Server Error",
  "path": "/glasses/delete/4",
  "status": 500,
  "timestamp": "2023-03-31T21:32:54.096+00:00"
}

```

Figure 15. Test 7 for Deleting a Glasses Data by Id in the DB using a Json (Httpie) (Will be improved)

```

C:\Windows\System32>http GET :8080/glasses/find/1
HTTP/1.1 200
Connection: keep-alive
Content-Type: application/json
Date: Fri, 31 Mar 2023 21:16:48 GMT
Keep-Alive: timeout=60
Transfer-Encoding: chunked

{
  "glassesCode": "61d7c91f-dacc-493d-967d-64a1e4cda862",
  "imageUrl": "https://www.kits.ca/cart/images/frames/items/5/50/50000001564_IMG.jpg",
  "name": "Ray-Ban RX5356",
  "shape": "Square"
}

```

Figure 15. Test 8 for Finding a Glasses Data by Id in the DB using a Json (Httpie)

More tests will be added as more entities and APIs are developed.

- d) As we have learned, REST, as the name suggests, stands for Representational State Transfer. REST principles are a set of protocols for designing web services using HTTP and standards to enable communication between the systems. Our implementation for Deluxe vision follows REST principles as we have done the following:
  - We implemented authorization and authentication at the server end to protect resources.
  - To make it easier for our clients to interact with the API, We used a consistent URI structure for the server.
  - Caching implemented to improve performance and reduce the number of requests.
  - Statelessness: Make sure that each request contains all the necessary information to complete the request.

By following REST we have created a backend that is easy, secure and scalable in the future

## 1.7 Demo

The quality attributes (security, performance, and scalability) that this system supports:

### Security:

Both Angular and Spring Boot have built-in security features. Spring Boot has Spring Security, which can be easily integrated to provide authentication and authorization mechanisms. Angular also has security features that help prevent cross-site scripting (XSS) and other injection attacks. While the security measures provided by these frameworks can help protect the application, it is essential to ensure proper implementation and customization of the security features to fit the specific requirements of the e-commerce website.

### Performance:

Angular is a popular framework for creating responsive and fast single-page



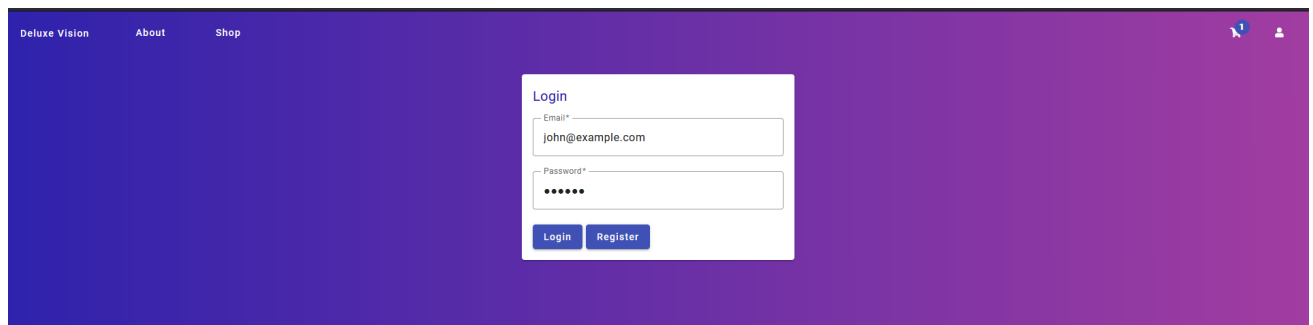
applications (SPAs). It uses ahead-of-time (AOT) compilation and lazy loading to optimize the performance of the application. Spring Boot, on the other hand, is a lightweight and efficient framework that simplifies Java development and can be configured to optimize performance. By leveraging these features, the Deluxe-Vision application should provide a good user experience and quick response times.

### Scalability:

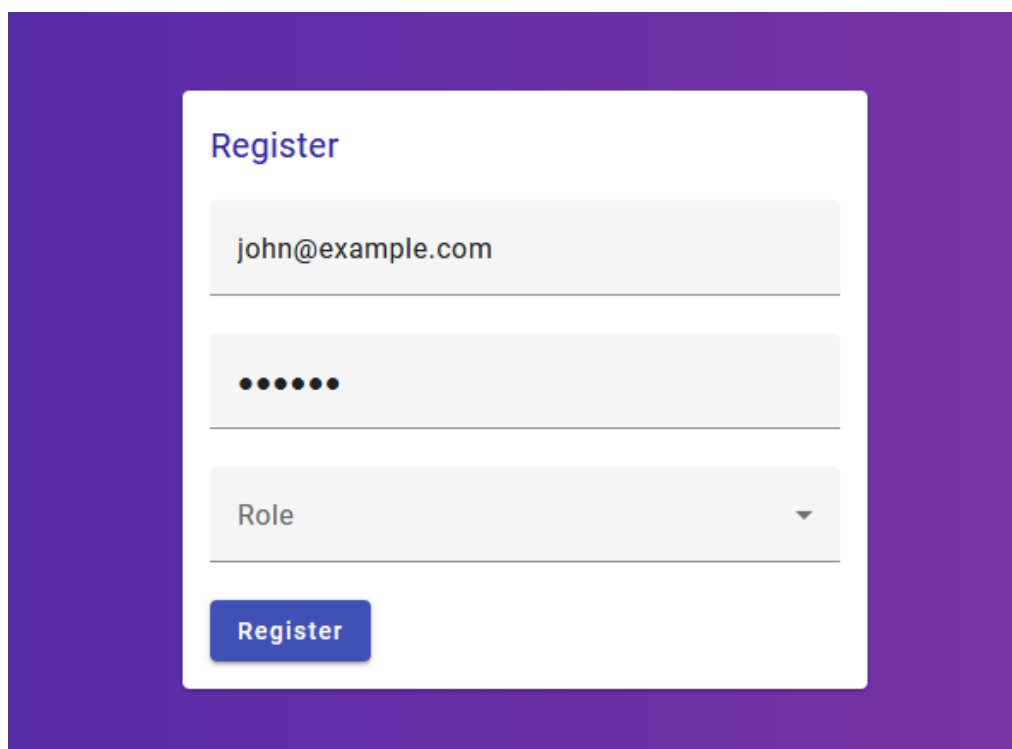
Scalability is crucial for an e-commerce website, as it may need to handle a growing number of users and transactions over time. Spring Boot applications can be easily scaled by deploying them in clustered environments, using load balancing, and leveraging microservices architecture. Angular applications can also scale well due to their modular architecture and efficient resource usage. By following best practices for both Angular and Spring Boot, the Deluxe-Vision application should be able to scale to accommodate a growing user base and increased traffic.

### b) Web Design:

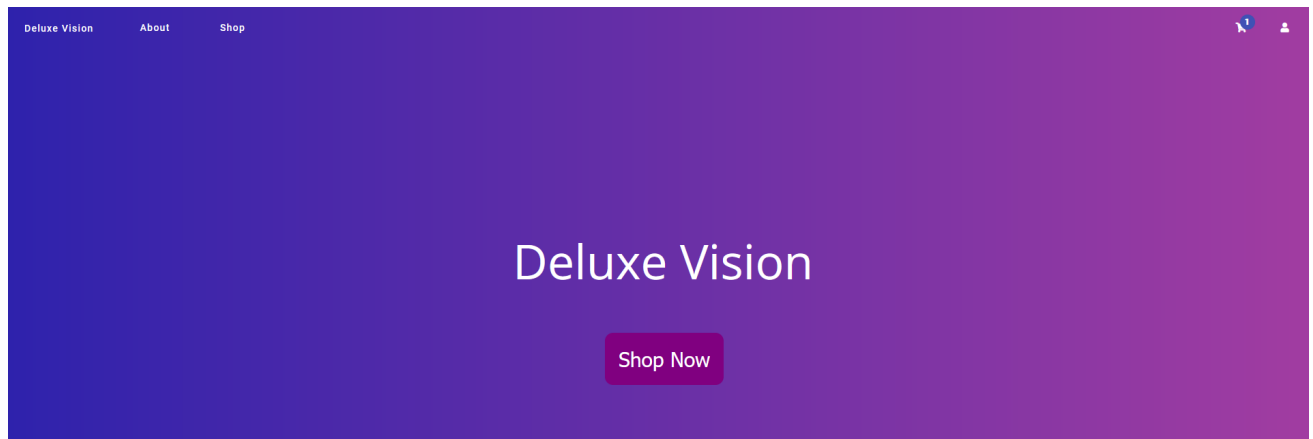
#### Login Page:

A screenshot of a web application's login page. The page has a dark purple gradient background. At the top, there is a navigation bar with the text "Deluxe Vision", "About", and "Shop" on the left, and a user profile icon on the right. In the center, there is a white login form titled "Login". The form contains two input fields: "Email\*" with the value "john@example.com" and "Password\*" with masked characters "•••••". Below the input fields are two buttons: "Login" and "Register".

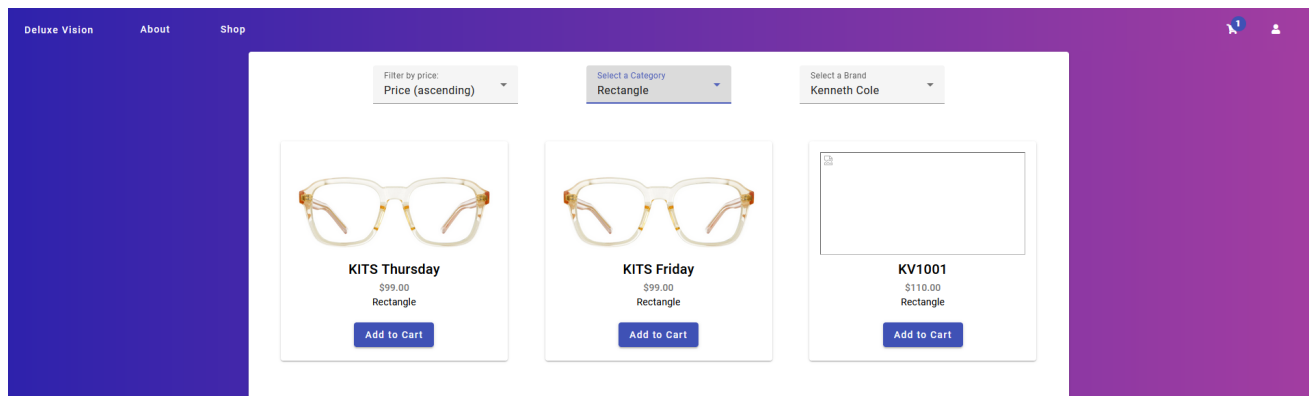
#### Register Page:

A screenshot of a web application's register page. The page has a dark purple gradient background. In the center, there is a white register form titled "Register". The form contains three input fields: "Email" with the value "john@example.com", "Password" with masked characters "•••••", and "Role" with a dropdown arrow. Below the input fields is a blue "Register" button.

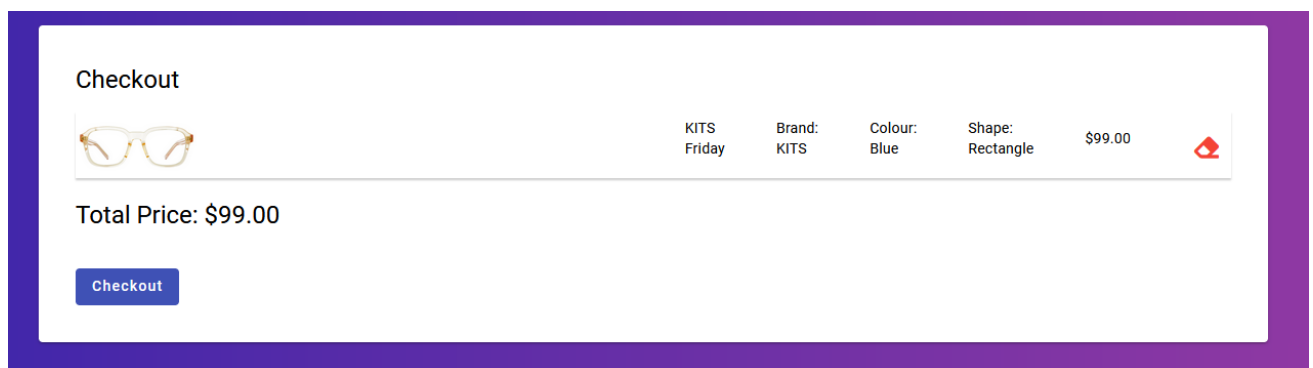
## Landing Page:



## Shop Page:



## Checkout Page:



These are all the main pages of the website. The user starts off at the landing page where they are greeted by a button to start shopping. From the shopping page they can browse different glasses and add them to their shopping cart. (The images are being extrapolated still). They can filter different brands and categories for their glasses. From there they can checkout their items. Checkout only occurs if the user is logged in. If the user does not have an account they can use register to register an account into the database and go back to login.

c)

- Host the website on the cloud
- Setup any final controllers or API's for the backend
- Create report
- Implement usable data

d) The client side implementation used 5 different technologies:

Our front-end development utilizes several key technologies to create a robust, responsive, and scalable web application that is user-friendly and visually appealing.

Angular is our front-end development framework of choice. It offers a modular architecture that enables us to create reusable components. They can be easily incorporated into multiple applications. This makes it easier to develop and maintain large-scale applications also also improves our overall performance and user experience.

We also used Typescript. TypeScript is used to build strongly-typed, object-oriented programming with advanced features that ensure code is more maintainable and error-free. TypeScript provides us with compile-time checking and other features that help catch bugs before they make it to production.

Angular Material is our UI component library, providing customizable pre-built UI components such as buttons, menus, forms, and other elements. By using Angular Material, we can create a consistent, responsive, and visually appealing user interface. Angular Material also offers accessibility and internationalization features, making our application more user-friendly for a wider range of users.

FontAwesome is our library of scalable vector icons that can be easily customized and used in web applications. By using FontAwesome, we can add icons and other visual elements to our application, making it more visually appealing and user-friendly.

LocalStorage is a built-in feature of modern web browsers that allows web applications to store data on the client-side. We utilize LocalStorage to store data such as the user's shopping cart and preferences, reducing the load on our server and improving the application's speed and responsiveness. By relying on LocalStorage, we can offer a faster, more responsive experience to our users, while also making our application more reliable and efficient.

f) Strengths:

- Our system provides a user friendly experience and interface for customer to browse and purchase products
- The use of angular and material allows for easy component reuse
- The LocalStorage implementation allows for storing data on the client-side enhance the applications speed and responsiveness
- The use of a REST API ensures secure and efficient data transfer between the client-side and server-side.

Weaknesses:

- Currently, the system lacks the ability to track orders and provide order history for customers.
- The system could benefit from the addition of search and filter functionality to enhance the browsing experience.
- The use of LocalStorage limits the amount of data that can be stored, which could become an issue as the application scales.

I enjoyed the process of designing and implementing the user interface using Angular and Angular Material. It allowed for easy and efficient development of visually appealing components.

What I enjoyed the least: I found that incorporating the CSS was challenging at times, but it was a valuable learning experience.

Technology not explored in class: I would have liked to learn more about serverless architecture and its implementation in web applications.