4413 F24 Team project report

Team composition

The description of our team is as follows:

- Team name: Cipher
- Team member 1: Vincent Tran (219058460)
- Team member 2: Fouad Shalaby (218638692)
- Team member 3: Oluwamayowa Isaac Ibidun (219070861)
- Team member 4: Asadullah Usmani (217010372)

Vincent's contributions
Backend implementation (Models,Repos,Services,Controllers)
Web deployment
JS scripts for front end (login functionality & enforcing user roles)
Design description
Advanced and distinguished features
Back end implementation description
Deployment efforts
V.T
Fouad's contributions
Backend implementation(Models,Repos,Services,Controllers)
JS scripts for cart(guest cart/user cart/transfering)
JS script for search functionality
JS script for registration
System Design/Structure Architecture
Database Architecture Structure/Creation
High-level Arch & structural description/Architectural Diagram
F.S
Isaac's contributions
Entire frontend design and development
Integration of frontend and backend for catalog, product, profile, checkout and admin pages.
I.I.
Asadullah contributions
Half of checkout.html, Introduction, Conclusion, Frontend description, use case and sequence diagrams.
AU

Table of content

Introduction

Our e-commerce platform, Elysium, is built to present a broad selection of electronics while offering a smooth and engaging shopping experience. Customers can browse through the product catalog, apply filters, and access in-depth details about specific items. The site is designed with easy navigation, interactive product features, and detailed pages for electronics. By focusing on responsive design, Elysium ensures a consistent user experience across various devices, all while maintaining an attractive design. Standout features include a live search bar, options to filter by category or brand, and a product details page that dynamically displays relevant information.

The platform follows the Model-View-Controller (MVC) framework, which helps keep the code well-organized and easier to maintain. The structure and styling are built using HTML and CSS, with responsive techniques ensuring compatibility across different screens. JavaScript allows the interactive elements, with its modular organization enabling clear and manageable code. Features like localStorage are used to handle seamless data sharing between pages, enhancing the site's dynamic capabilities. The user interface is further enhanced by tools like Google Fonts and Ionicons, while a consistent CSS framework simplifies styling. JavaScript event handling was employed to enable responsive product navigation and selection, and the use of reusable components streamlined development while fostering team collaboration.

The design's primary strength lies in its modularity and usability, making the project easy to extend and maintain. The interface is visually appealing, with consistent styling and responsive layouts. User interaction is intuitive, thanks to the dynamic product selection and filtering system. However, a notable weakness is the reliance on local storage for data transfer, which limits functionality if the browser's storage is cleared or disabled. Additionally, while the design is efficient for small datasets, scalability might require backend integration for dynamic data fetching and management. Enhancing security measures and optimizing performance for large product catalogs are areas for potential improvement.

Architecture Description

high-level overview

The Elysium system is built on modern architectural principles to ensure scalability, maintainability, and flexibility. Its architecture includes the combination of different design patterns, including the use of Client-Server, Repository Pattern, Layered Architecture, Service-Oriented Architecture, Model-View-Controller, and Event-Driven Architecture. These are design patterns intended and chosen to meet the challenges an e-commerce site brings and optimize its performance, security, and end-user experience.

This report gives a detailed description of each of the architectural styles in Elysium and their effects on the overall structure and functionality of the platform.

1. Client-Server Architecture

Overview:

Elysium was built to follow a Client-Server architecture, the C**lient** (*front end*) interacts with the **Server** (*backend)* through RESTful APIs. This is a common reference architecture style for many web applications, in particular, those that require distinct separation between the user interface and business logic

i.Client-Side:

The front end is responsible for handling all the user interface and user interactions. It is developed using the standard web technologies, including HTML, CSS, and JavaScript. The client side is designed and built to be lightweight, ensuring fast load times and efficient user interaction. It communicates with the server through HTTP requests and processes responses asynchronously

- **Key Components:**
    - **Login Page:** The user logs into the website through the login, which handles authentication
    - **Product Page:** Showcases all the products for the user to view, add them to their cart, and proceed to checkout
    - **Checkout page:** The user views all their selected products, review cart, and the overall checkout process, and proceed to checkout

ii. Server-Side

The backend server handles all the business logic, data processing, and database interactions. It is built using **Spring Boot,** providing a powerful and robust framework for developing and creating RESTful APIs that the client side interacts with:

- **Key Components:**
    - **Controllers:** Handle incoming HTTP requests and map or align them to appropriate service methods (*e.g CartController, User Controller)*

    - **Services:** Service Classes (*e.g. OrderService, CartService)* process the core logic, such as managing user sessions, checking out orders, and calculating total prices
    - **Repositories:** Repositories (*e.g. ProductRepository, CartRepository)* manage interactions and access the database

iii. Benefits:

- **Modularity;** the client and the server are distinct, each responsible for different parts of the system which separates the concerns.
- **Scalability:** The system can be scaled by improving the frontend independently of the backend, or vice versa
- **Security:** the backend can be secured independently from the client, and confidential data can be protected through server-side authentication and authorization mechanisms.

2. **Repository Style**

Overview:

The **Repository Style** is used in the backend to abstract data access and business logic since it centralizes the database. It provides a layer of abstraction between the domain and data mapping, allowing the system to be more maintainable and testable.

i. Implementation:
In Elysium, **JPA repositories** (*e.g. ProductRepository, UserRepository, CartRepository)* are used to interact with the database. These repositories provide built-in methods for *CRUD* operations and custom query methods, such as searching for products by name, price, or category

- **Example:** ***ProductRepository*** provides methods like ***findByCategoryName, findByBrand,*** and ***findByPriceLessThan***, allowing the business logic layer to interact with the database without directly managing SQL queries**.**

ii. Benefits:

- **Abstraction:** The Repository Style abstracts the complexities of database interactions from the service layer
- **Maintainability:** it makes the system easier to maintain by centralizing data access logic in one place
- **Testability:** Repositories can be mocked during testing, enabling easier unit testing of the service layer

3. **Layered Architecture**

Overview:

The System follows a **Layered Architecture** that breaks down the system into four distinct layers. This structure allows for improved maintainability and scalability by isolating different aspects of the system into specialized layers. Three main Layers Client layer,Service layer which is broken down into two other layers(Supervisor & Core) and Data layer:

i. Layers:

1. **Client Layer**(*Frontend*)**:** The front end (client-side) that handles user interaction, loading the UI, and sending requests to the backend API
2. **Supervisor Layer**(*Controller)***:** The backend controllers (*e.g. CartController, UserController)* receive HTTP requests, validate input, and assign business logic to the *Core Layer.*
3. **Core Layer(***Service/Business***):** The services (*e.g. CartService, OrderService, ProductSerivce)* encapsulate the core business logic, such as calculating order totals, managing cart items, and handling product inventory
4. **Data Layer**(*Repositories*): The repositories (*e.g. ProductRepository, CartItemRepository)* interact with the database and manage data persistence. They provide an abstraction layer over the underlying database operations.

ii. Benefits:

- **Separation of Concerns:** Each layer will have a single responsibility; this makes the code base easier to understand and change.
- **Modularity:** Changes in one layer, for example, a change in the frontend UI, do not affect other layers, such as business logic or data access.
- **Scalability:** Layers or components can be added without affecting the core functionality of the system.

4. **SOA - Service-Oriented Architecture**

Overview:

The system has followed the SOA architecture, basically aimed at developing modular services representing portions of the business processes for flexibility in the system with easy scaling.

i. Implementation:

Each key business process should go under encapsulation by a respective service such as:

- **CartService**: For handling all user current carts (items addition/ removal and overall calculation);
- **OrderService:** Handles ordering checkout - payment processing with inventory;
- **ProductService**: Covers product information including their prices, inventories, etc.;
- **UserService**: This handles user sign-up, login, and authentication.

These services interact with each other through well-defined APIs and are independent, which means the system can grow by adding new services without disrupting the already existing ones.

ii. Benefits:

- **Loose Coupling**: Services are independent, interacting only through predefined interfaces.
- **Scalability**: New services can be added as the system grows, enabling the platform to expand its functionality without major architectural changes.
- **Flexibility**: Services can be updated or replaced without affecting other services.

5. **Model-View-Controller (MVC)**

Overview:

Even though the system is much more than a typical web application, it still applies to the Model-View-Controller pattern. This pattern will be very useful in web applications where the user interface needs to be separated from the application logic and data.

i. Implementation:

- **Model**: It is the representation of the data. In this case, examples include Product, Cart, and User. Core business objects reside here.
- **View**: The frontend interface, HTML and JavaScript, is what the user interacts with; it shows the data to the user.
- **Controller**: Backend controllers, such as *ProductController* and *OrderController*, process user requests, interact with services, and return the appropriate response.

ii. Benefits:

- **Separation of Concerns**: Data, presentation, and logic stay in their respective places, and thus the system is easier to maintain.
- **Ease of Maintenance**: Changes either on the UI (view) or business logic (controller) can be made independently.
- **Testability**: MVC components can be tested individually, improving the system's testability.

6. **Event-Driven Architecture**

Overview:

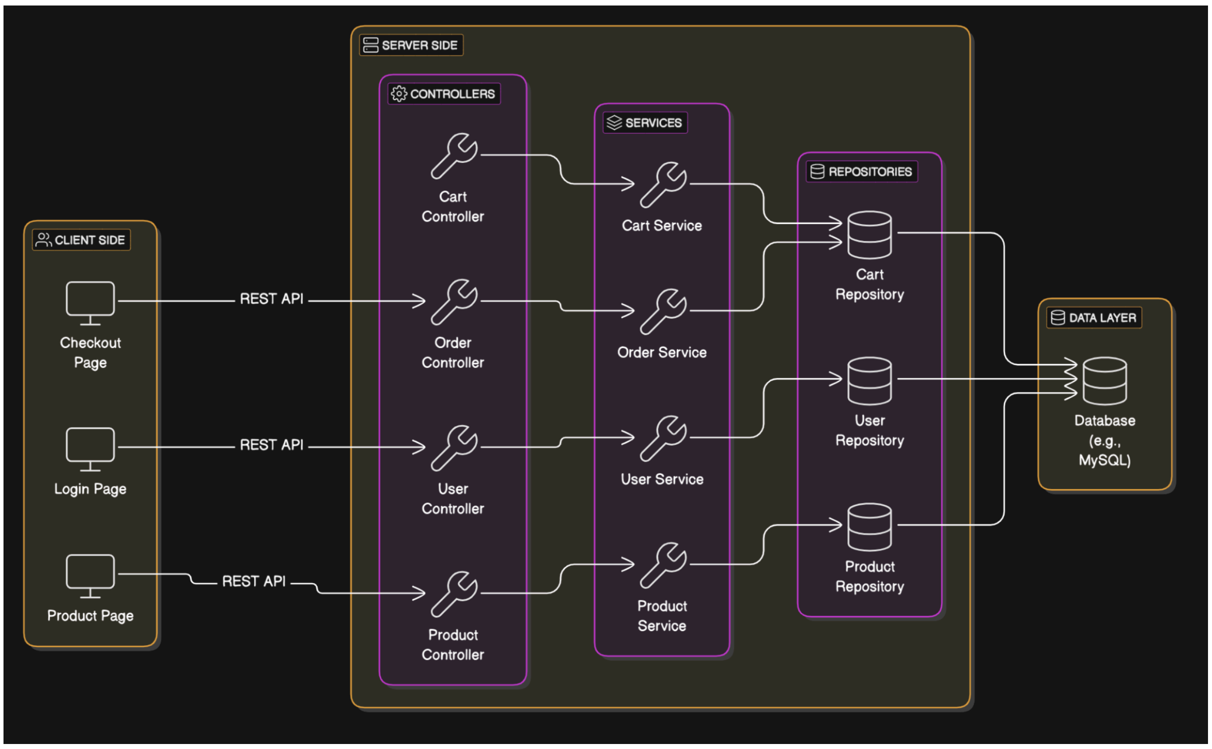
Event-driven architecture can be implemented at a later stage of maturity when the system needs to decouple services and allow different parts of the system to communicate asynchronously. The system is not event-driven for the most part, but it is designed in such a way that it can accommodate event-driven patterns in the future.

i. *Implementation*:
For instance, on an order being placed, it might trigger an event to update inventories, send an email to the user, and alert other subsystems of the transaction. Services can listen to these events and respond as they see fit.

ii. Benefits:

- **Decoupling**: System components can be loosely coupled; thus, it provides better flexibility and scalability.
- **Scalability**: The system can asynchronously support high-volume operations; hence, more scalable.
- **Asynchronous Processing**: Long-running processes, such as sending confirmation emails or processing payments, can be offloaded asynchronously to improve system responsiveness.

7. **Architecture Diagram**



Conclusion:
The Elysium system is architected using a combination of Client-Server, Repository Pattern, Layered Architecture, Service-Oriented Architecture (SOA), Model-View-Controller (MVC), and Event-Driven Architecture. This hybrid approach will ensure that the system is modular, scalable, maintainable, and flexible.

By applying these architectural styles, Elysium will be able to manage user interactions, business logic, and data access effectively, while it grows and scales easily for the future. Each component of the system plays a critical role in ensuring that the platform can meet the demands of an evolving e-commerce landscape.

Use cases

Use Case Name: Product Browsing and Filtering

Scope: Online Shopping Website

Level: User-Focused Goal

Primary Actor: Customer

Stakeholders and Objectives:

1. **Customers**: Try to search for products by searching and filtering by brand, category, and price. E-Commerce Platform: This increases the sales by improving user experience through effective product selection.

Preconditions:

- The customer accesses the index.html page.
- The website is up and the products can be displayed.

Main Success Scenario (or Basic Flow):

1. The customer launches the e-commerce website by opening the index.html page.
2. They interact with filtering options to refine the product list, choosing from Brand and Category. Optionally, results may be sorted by Price or by Name.
3. Additionally, the customers may choose to sort the results by Price (low to high or high to low) or Name (A-Z or Z-A).
4. The website dynamically updates and displays the filtered product list based on the customer's selections.

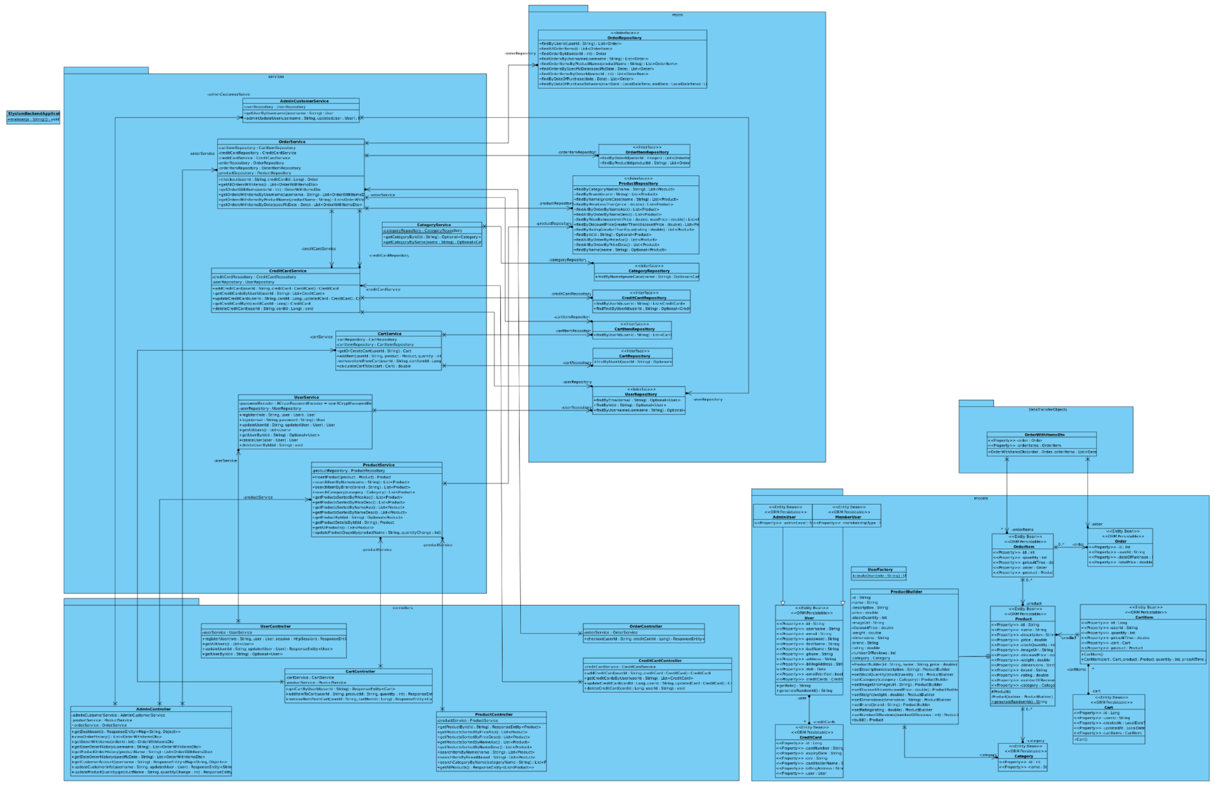
Admin Perspective:

The *Admin* actor supports the functionality by managing the backend:

- Admins maintain the catalog by adding or removing products and updating inventory.
- They process user orders by managing statuses (e.g., pending, shipped, delivered).

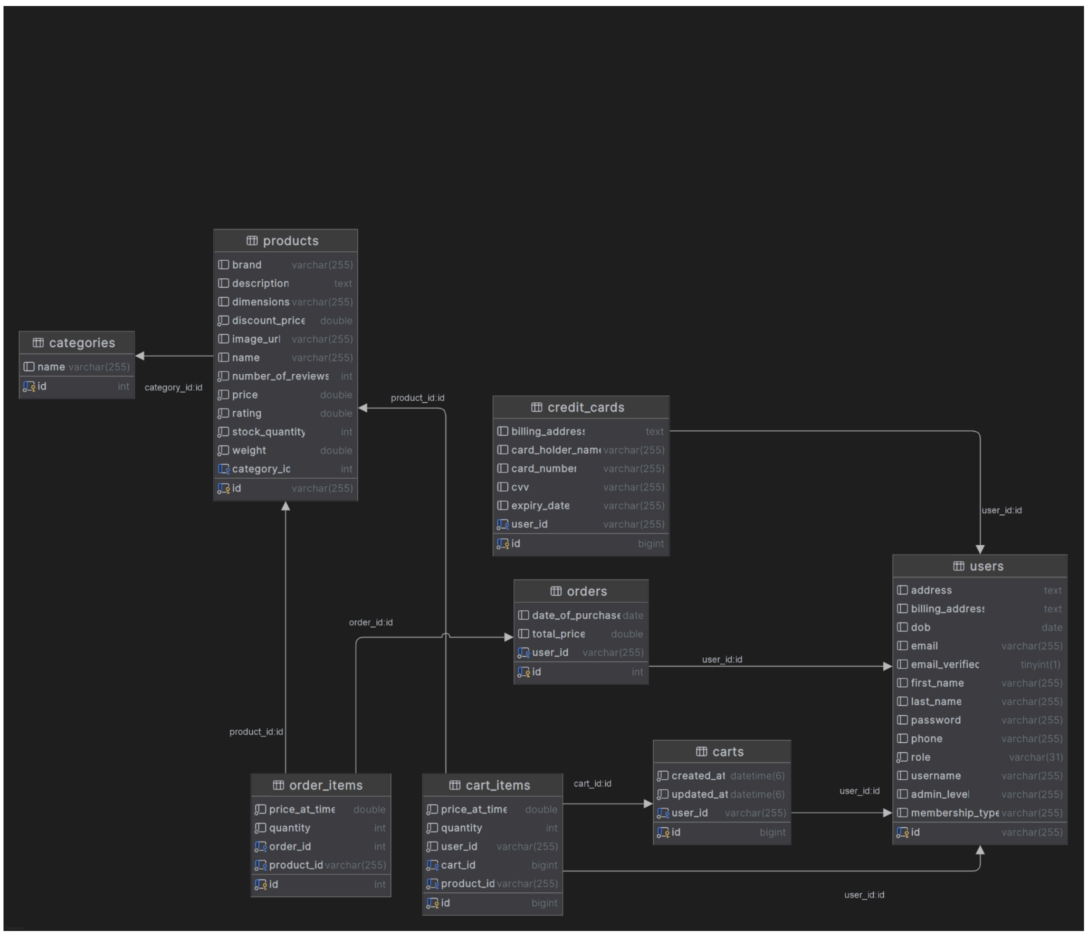
Design description

UML class diagram

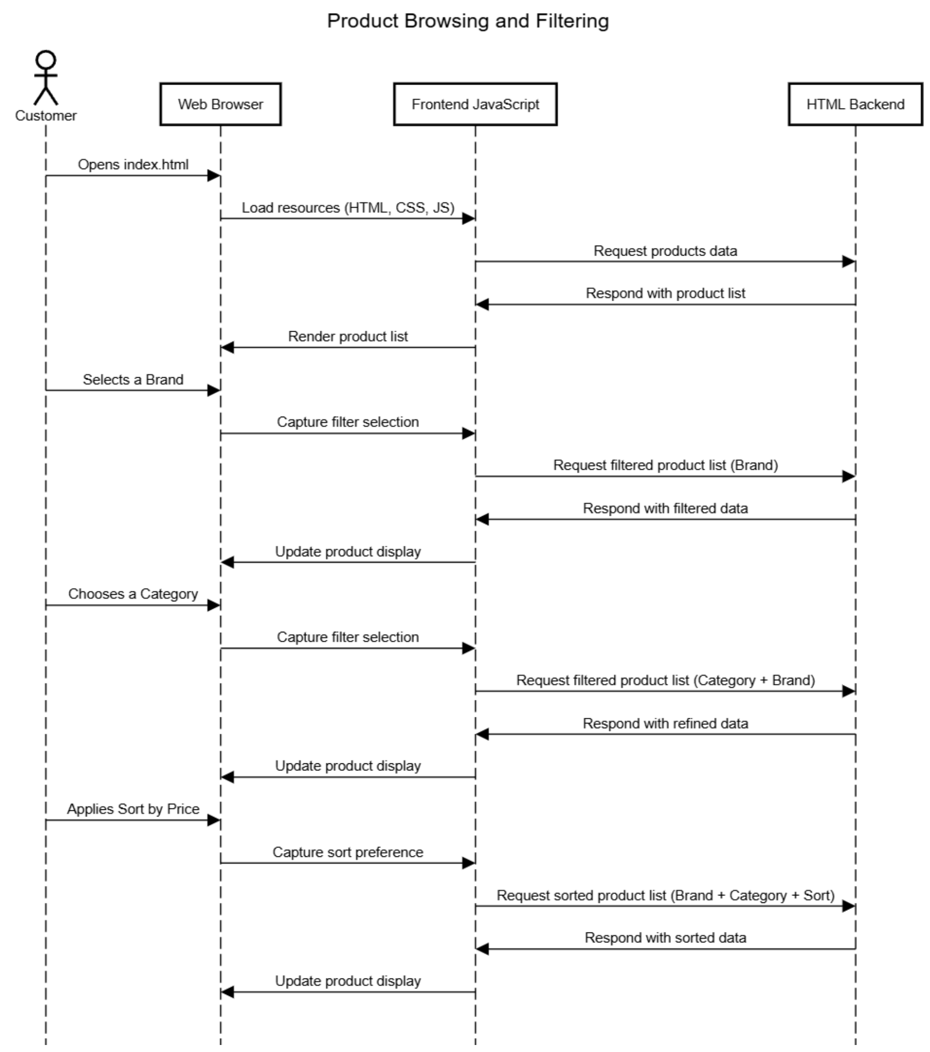


(higher resolution diagram in project zip)

database schema



Sequence diagram



introduction

Elysium Model-View-Controller (MVC) pattern for clear separation of concerns and the Data Access Object (DAO) pattern for database interactions through the use of java spring boot, enabling modularity and scalability.

MVC

model

The model represents the data and business logic of the application. For example, the entity “User” is implemented as Java classes annotated with JPA (@Entity annotation)

view

The view is responsible for presenting data to the user. This is implemented using plain HTML in combination with javascript and CSS.

controller

The controller manages user requests, processes them, and returns appropriate views or data. In our application, controllers are annotated with @RestController or @Controller to handle HTTP requests.

DAO

DAO is another design pattern we used in order to enhance our application. This layer handles database operations and abstracts persistence logic. It interacts directly with the database through Spring Data JPA repositories, ensuring that data manipulation code is separate from business logic.

Design decisions

- Spring Boot simplifies the project configuration and deployment, as it configures java beans automatically based on dependencies detected in the class path and allows us to package our application with an embedded server
- MVC design pattern helps organize our web application and makes it more manageable
- DAO design pattern ensures that database-specific code is encapsulated, allowing changes to the database without affecting other layers

trade-offs

- heavy reliance on Spring Boot and JPA results in limited flexibility if switching frameworks or databases in the future
- MVC, while making maintaining the application easier, introduces complexity due to multiple layers of abstraction.
- Using a deep layered results in some overhead.
    - For DAO, the application must go through multiple abstraction layers in order to access the database. It must go through the controller, service and then finally the DAO layer before reaching the database

Advanced and distinguished Features (Beyond the requirements)

1. **Factory pattern**

- Encapsulates the logic required to create instances of Admin and Customer. Instead of manually instantiating these objects throughout the application, the factory provides a single point of creation
- This pattern also allows the client code to depend on the User parent class instead and not the not the specific implementations (Admin, Customer). This reduces coupling and increases flexibility.

2. **The product has inventory information**

3. **Users can see their account pages with their personal information and purchase history**
4. **Administrators can maintain customer account pages with customer info, purchase history, etc**
5. **Administrators can maintain inventory, to change the quantity of products (add inventory)**
6. **Application enforces user roles**

- a user can only access admin pages/tools only if they are signed into an account that has the role “admin”
- a user can only access member pages/tools only if they are signed into an account that has the role “member”

Implementation

Front end Implementation

Our front-end implementation includes the layout structure, product catalog, responsive design, technology integration, and the aesthetics and the branding. The layout structure includes the logo (which is called Elysium), the search bar to search for items, the sign in button to sign in with the users account and the cart which has all of the items the users add to their virtual cart.

The product catalog includes dropdown menus which allows the users to categorize and sort them by alphabetical order, low to high price and vice versa. The products on the website also include the image, price and an “Add to Cart” feature as well. The images of the products have high pixels and have a high response time when the user clicks on any item.

The aesthetics of the brand and website stands out to attract customers and to shop online at Elysium consistently. The styles of the button are big and easy to click with the quick response time. The website is also easy to navigate.

Back end Implementation

For the back end, our team explored several different technologies such as Servlets as we learned in class, Spring Boot, and Node.js. Ultimately our team chose to use Spring Boot due to its modularity, scalability and ease of implementation of the MVC and DAO design patterns.

1. Servlets

Java servlets, although we were all familiar with it due to the course, had a lot of drawbacks when it came to the project requirements. Servlets lack clear MVC architecture support as they handle both HTTP requests and responses, but also often include business logic and presentation logic within the same code. Thus encouraging a monolithic structure where the controller, view generation, and sometimes even data access are bundled in a single file. It also fails at providing an abstraction for the DAO design pattern, as it requires manual management of database connections, transaction handling, and integration with other business logic, making the code prone to bugs and harder to maintain.

2. Node.js

Similar to Java servlets, Node.js also had its downfalls when it came to the requirements of the project, but it also was not familiar amongst the team members. Since Node.js is a runtime and not a framework, it doesn't provide any built-in ways to implement the MVC architecture. Thus the user manually has to set up the MVC architecture, which can be time-consuming, error-prone, and hard to maintain. Although Node.js can handle Web API development, it lacks any built-in utilities like request validation, middleware, and structured routing. The user then has to manually set up routing, handle JSON payloads, and manage status codes, which can be prone to error and complication.

3. SpringBoot

In the end, our group chose to use Spring Boot due to its great features and ecosystem that support the project's requirements. Spring Boot, unlike the other options, naturally enforces the MVC design pattern. The controller is handled via @Controller annotation, the model is encapsulated as java objects, and the view is supported with HTML, JS, and CSS. This built-in framework ensures that business logic, presentation, and control are independent and maintainable. It has built-in tools for web API development, supporting RESTful APIs with @RestController, @RequestMapping, and @ResponseBody. This makes it so that building and managing the web API between the frontend and backend is straightforward and efficient. Finally, Spring Boot has built-in support for the DAO design pattern. It integrates with Spring Data JPA, making it easy to implement the DAO pattern. It provides repository interfaces like JpaRepository to encapsulate data access logic and dependency Injection ensures the DAO layer is independent and reusable across the application. Therefore, for all of these reasons, our group chose to use Spring Boot for our backend development.

Deployment efforts

Initially we tried to deploy our project either to docker or AWS (Amazon Web Services), but decided not as there were a few complications with both choices

docker

- our project is a fairly small one, so although docker's use of containers are fairly lightweight, we decided that the overhead that comes with it is unnecessary and that there are possibly better alternatives
- unlike deploying online, docker containers are stateless leading to data inside the container being lost when it stops running

AWS

- there was quite a large learning curve when it came to deploying our spring boot project onto aws as we had to configure services like EC2 or Elastic Beanstalk
- when we did try to deploy our project onto aws, we kept running into errors

Railway

In the end, our group chose to deploy our project onto a service called **Railway**. This is a cloud deployment platform that makes hosting applications and services easy, it allows developers to focus purely on the coding rather than managing a complex infrastructure that is compatible with web service deployment. It abstracts away many traditional deployment complexities, allowing you to deploy web applications, APIs, databases, and other services with minimal configuration. Nixpacks, a tool developed by railway, automatically generates buildpacks for applications and is what does most of the heavy lifting when we deploy our application. It simplifies deployment by automatically detecting your project's programming language and framework and creating an optimized build environment based on the detected type. Therefore we chose railway to deploy our application as it has automatic detection (no need to manually create docker files or other build configurations) and it simplifies the build for the user (handling Java version detection, dependency installation, and build steps for you).

Conclusion

For our project we adopted the Model-View-Controller (MVC) design pattern, which allows us to approach problem-solving in an organized and systematic way while enhancing code maintainability. Our project was built on a foundation of HTML, CSS, and JavaScript, with a modular file structure that helped us keep everything clear and well-organized. JavaScript's event handling, combined with localStorage, helped us create interactive features and enable data sharing across different pages. To ensure the interface worked smoothly on all devices, we applied responsive design principles and relied on reusable components. Tools like CSS frameworks, Google Fonts, and Ionicons added visual polish, which helped us deliver a functional and scalable ecommerce platform.

The design's main strengths included its flexibility for future updates, a responsive user interface, and a structure that could easily be extended. However, the reliance on client-side localStorage limited both scalability and security. While the system was perfectly suited for managing small datasets, scaling up to larger use cases would require the addition of backend support.

Reflection

- **What worked well:** Stable progress and on-time delivery were guaranteed by efficient teamwork and task distribution. The development process went more smoothly and team members were able to work freely without encountering any problems because of the team's commitment to modular design principles.
- **What went wrong:** Aligning individual contributions to a single design standard and coordinating efforts across time zones presented difficulties. It took longer than anticipated to troubleshoot cross-browser discrepancies and guarantee accessibility.
- **What we discovered:** The project improved our comprehension of how to use responsive design concepts and design patterns in practical situations. We learned a lot about working together on projects, using modular programming, and dealing with practical issues like browser compatibility and scalability.

- **Advantages and drawbacks of teamwork:** Collaboration fostered creativity by bringing together diverse perspectives, which improved both the concept and execution. It also reduced individual workloads through efficient task delegation. However, progress was occasionally slowed by varying skill levels and communication delays. One major challenge was maintaining consistent quality across all contributions. Despite these challenges, teamwork ultimately elevated the project's quality and provided a meaningful learning experience.