Group109

USCoffee
Software Architecture Document

Version <1.0>

Revision History

Date	**Version**	**Description**	**Author**
19/07/2023	1.0	First version of Software architecture document	Group 109

Table of Contents

Software Architecture Document

1. Introduction

- **Purpose:** The primary objective of the Software Architecture Document (SAD) is to furnish a comprehensive manual that delineates the architectural blueprint and arrangement of the software system under development. It aims to impart a clear comprehension of the essential components, their interactions, and the fundamental design principles governing the system. Serving as a crucial reference for all stakeholders involved in the project, including developers, designers, project managers, and clients, the document ensures a unified vision and understanding of the software's architecture.

- **Scope:** The scope of the Software Architecture Document (SAD) encompasses a detailed overview of the software system's architecture, which includes its various modules, components, and their interrelationships. This document outlines high-level design decisions, architectural patterns, and the technologies employed in the system's development. It covers both the functional and non-functional aspects of the software architecture, providing a comprehensive view of the system's structure and design considerations.

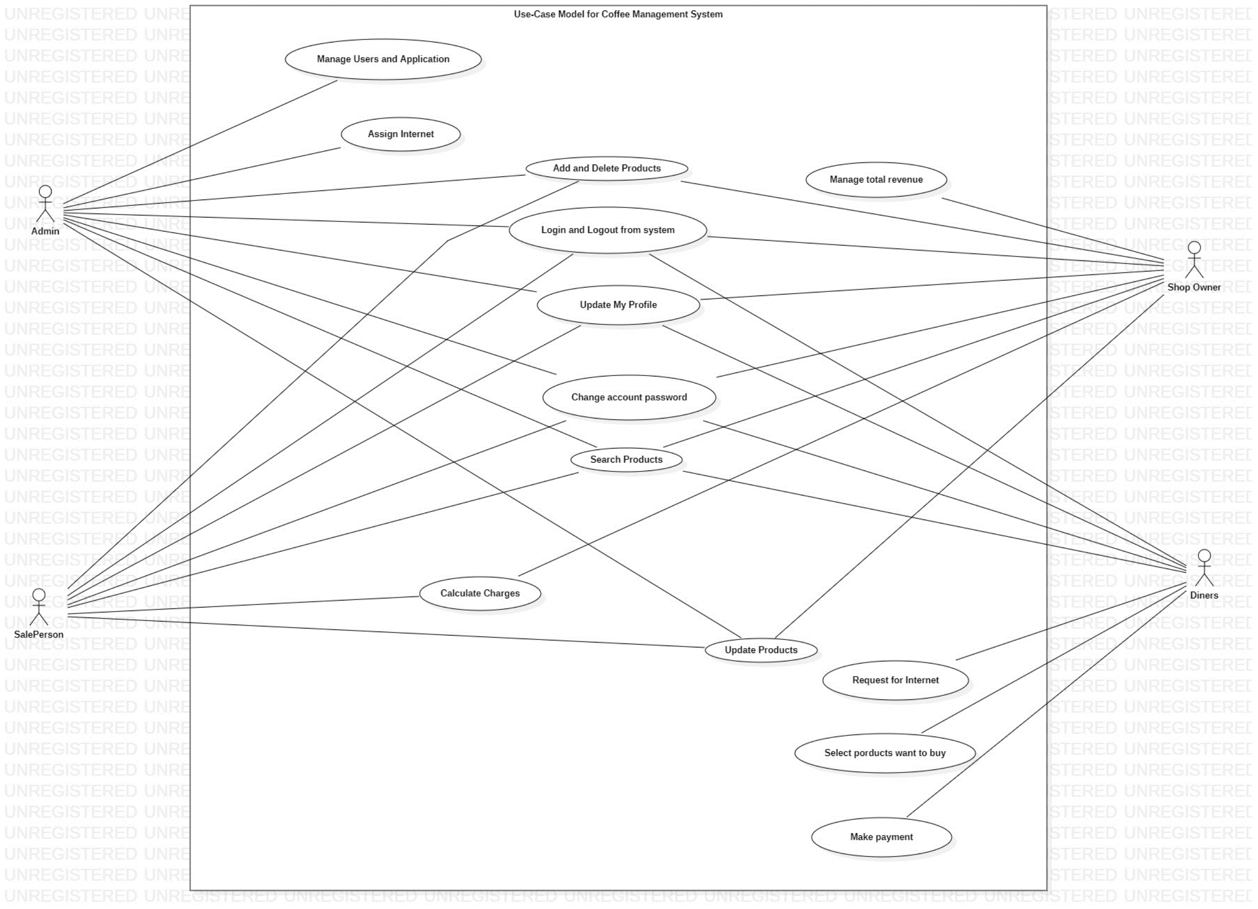
- **Definitions:** We will define important words and concepts related to the software system to make sure everyone involved understands them the same way.
- **Acronyms:** We will list the full meanings of any short forms used in the document to avoid confusion.
- **Abbreviations:** We will provide easy-to-find explanations for common short forms that are used in the software system.

2. Architectural Goals and Constraints

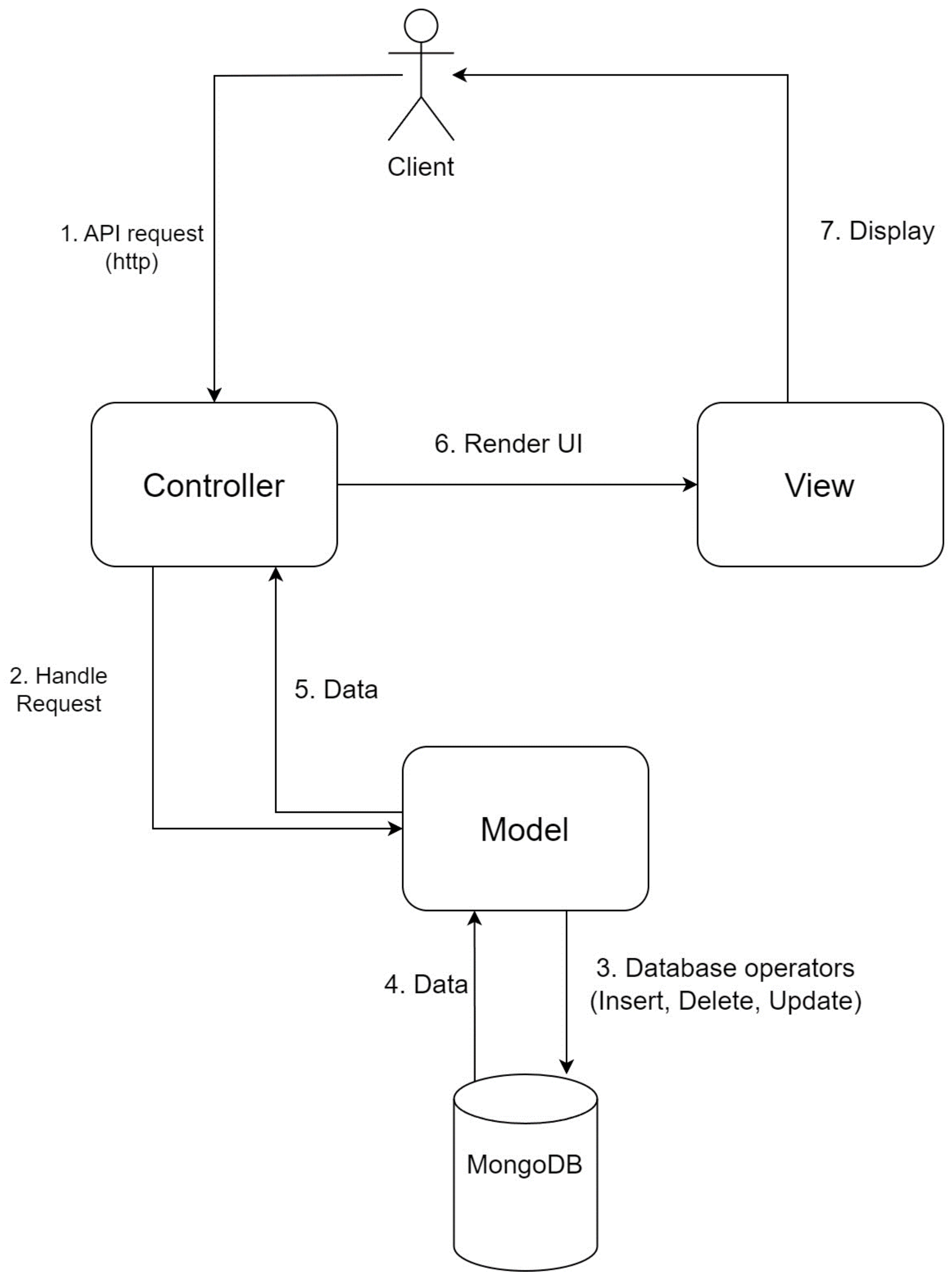
Some constraints requested for architecture:

- User's personal information should remain confidential and inaccessible to other users.
- All features request internet access to be used.

3. Use-Case Model



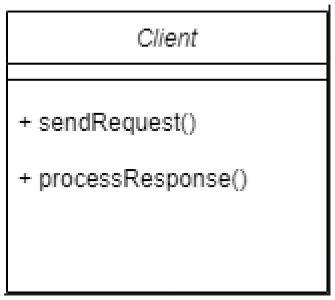
4. Logical View



4.1 Component: Client

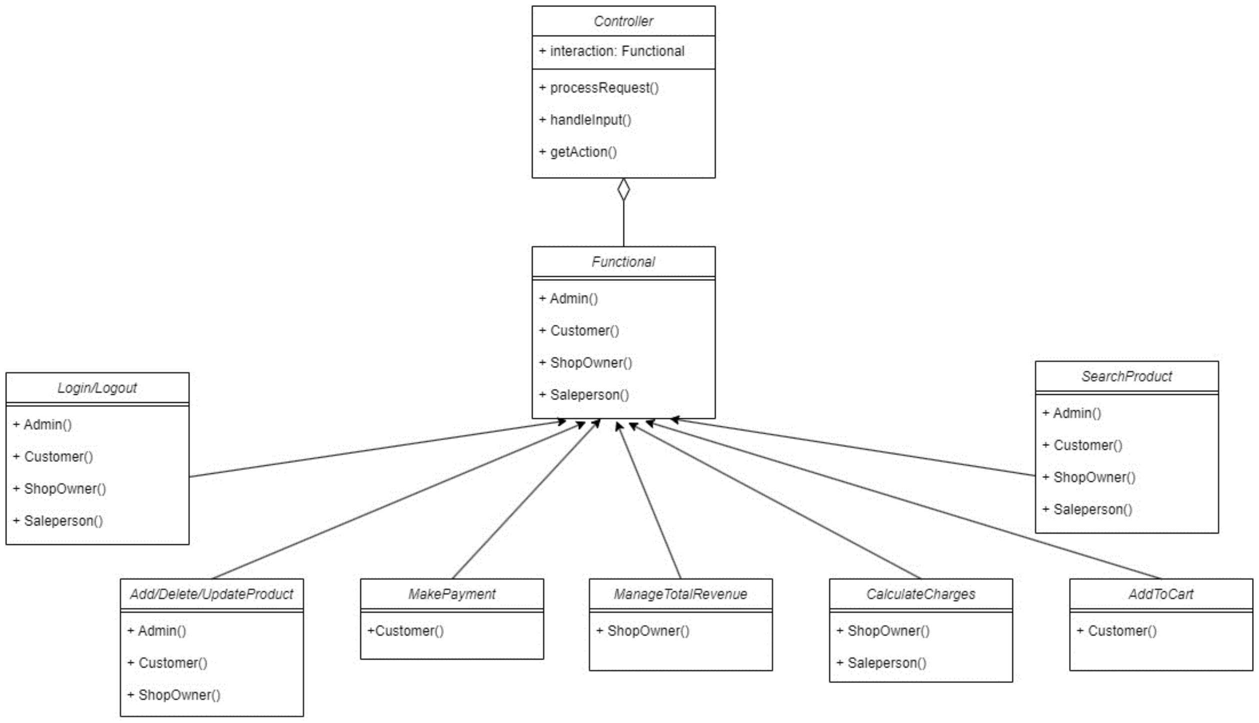
- **Client:** This represents the users' browsers or any client-side devices that interact with the website. The Client sends HTTP requests to the server to request data or perform actions. This class has 2 methods to process response and send HTTP request to controller

Functional



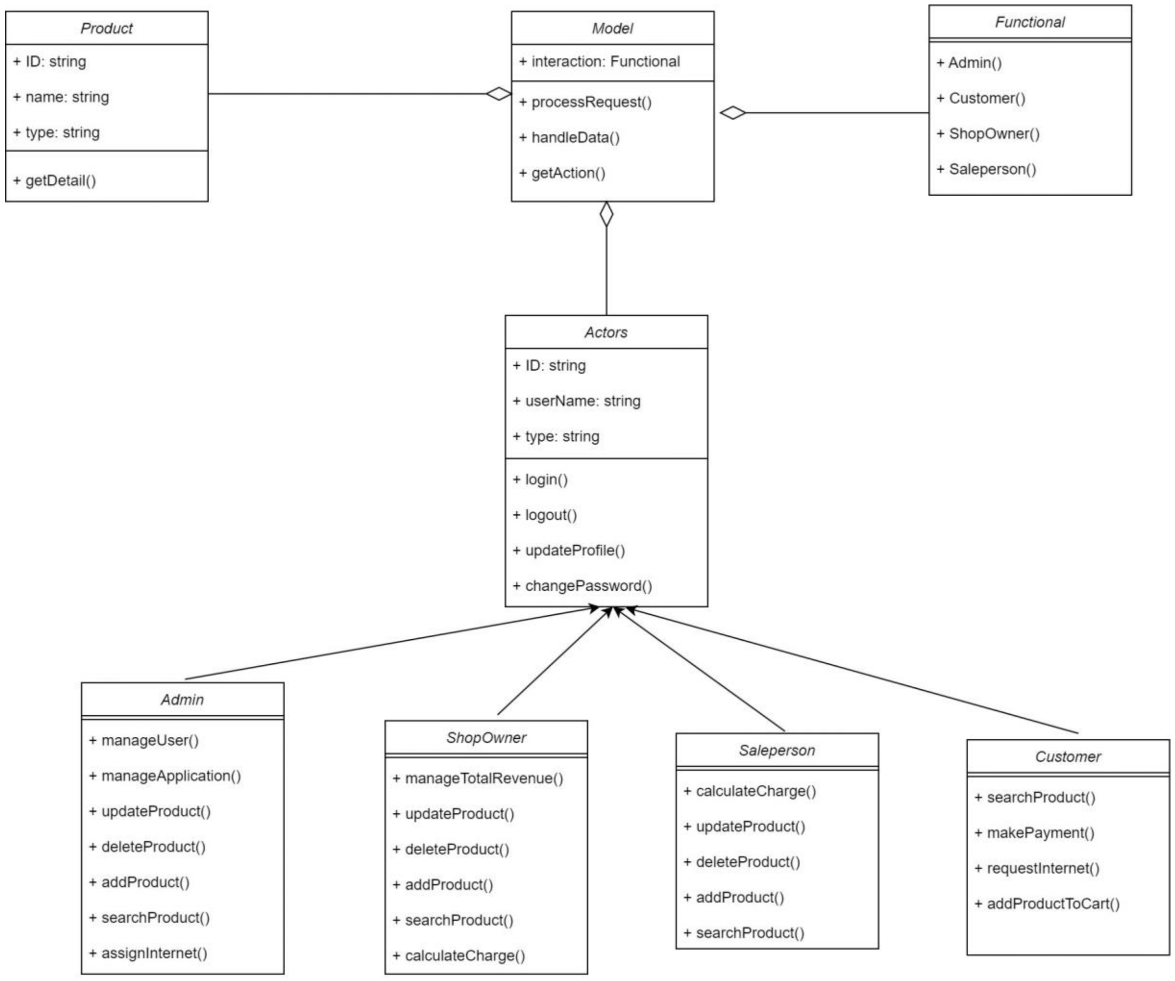
4.2 Component: Controller

- **Controller:** The Controller receives and handles the incoming HTTP requests from the Client. It acts as an intermediary between the Client and the Model. It processes the requests, fetches data from the Model, performs any necessary business logic, and sends the response back to the Client. The framework use for controller component is Next.js. In this diagram, the "Controller" class contains an attribute of type "Functional" and various methods for handling input, requests, and receiving actions from client
- **Functional:** An abstract class which has 4 functions representing 4 actors of the website. The classes below represent different use-cases, depending on each use-case inheriting from the "Functional" class with corresponding methods.

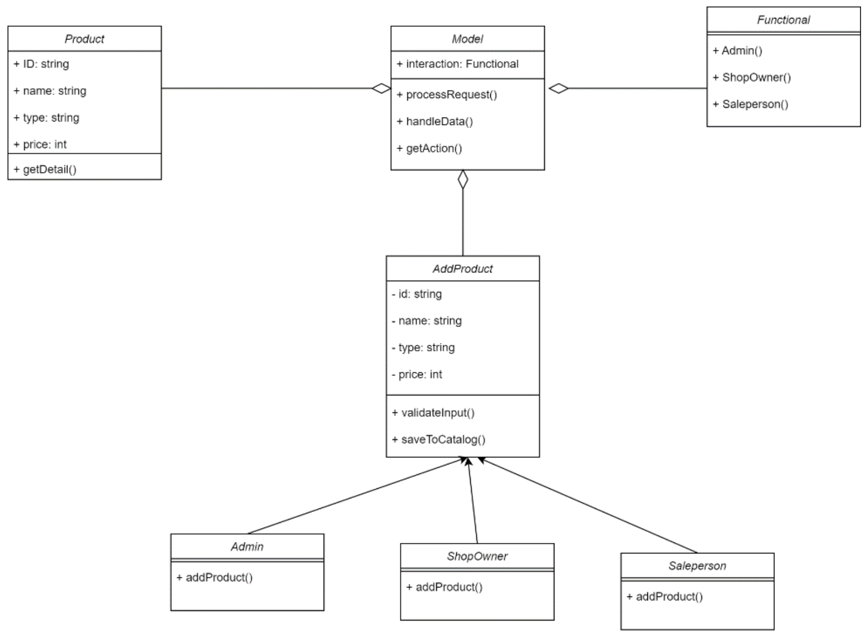


4.3 Component: Model

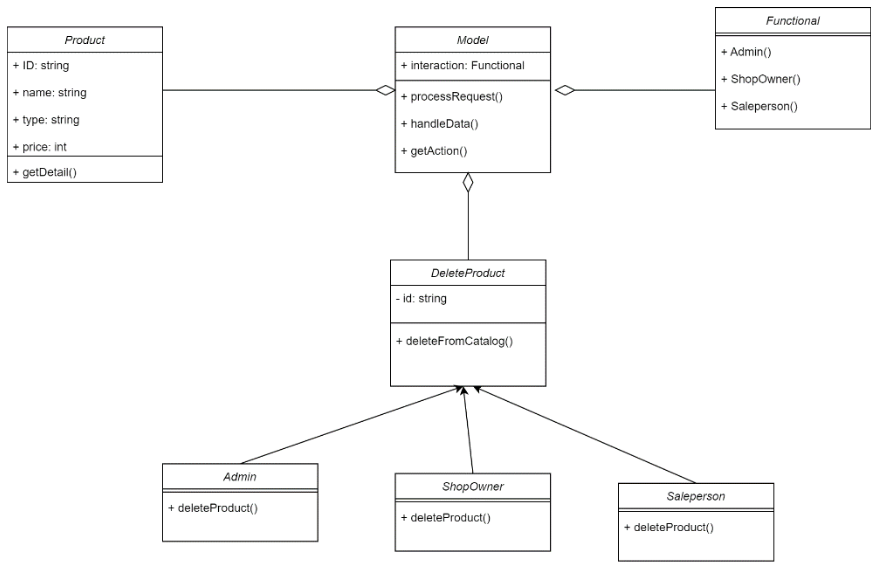
- **Model:** represents the data and the business logic of the application. It is responsible for interacting with the database (MongoDB) and performing various database operations like CRUD (Create, Read, Update, Delete) operations. The Model contains the application's data and state. Because the controller use Next.js, the Model will use MongoDB with Mongoose (an ORM library) for a MongoDB database. Similar to the "Controller" class, the "Model" class includes an attribute of type "Functional" and methods for processing requests, handling data, and receiving actions from "Controller"
- **Product:** This class includes attributes representing product details and a method called getDetail() to retrieve detailed information about the products.
- **Actors:** This class contains common attributes and general methods for users. The classes below represent different types of users, inheriting from the "Actors" class, and each user type has additional methods representing the services that each user type can use.



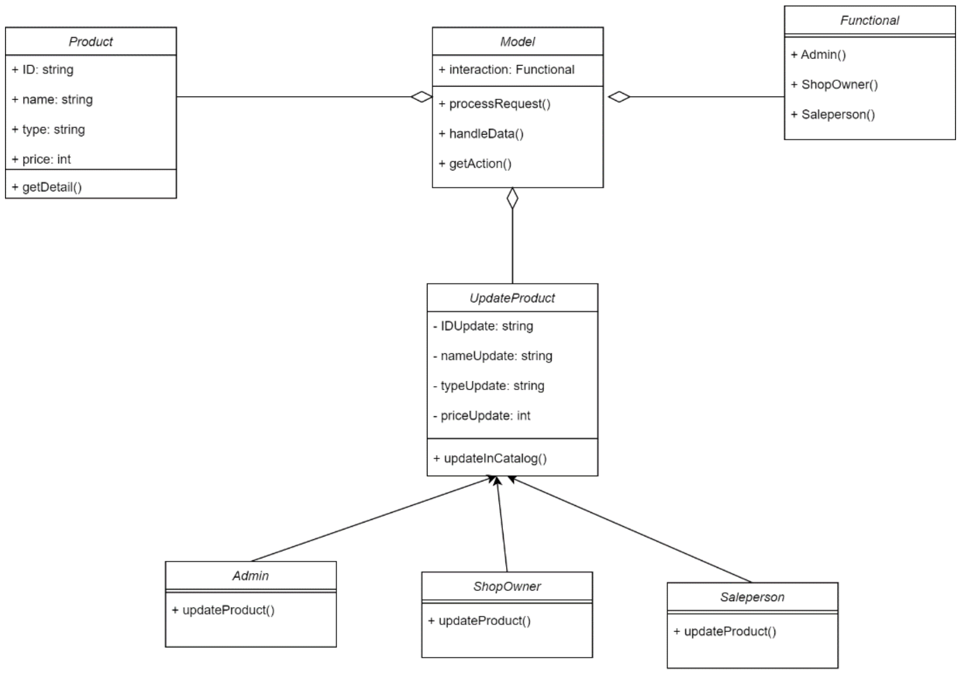
1. **Class diagram for Add Product**



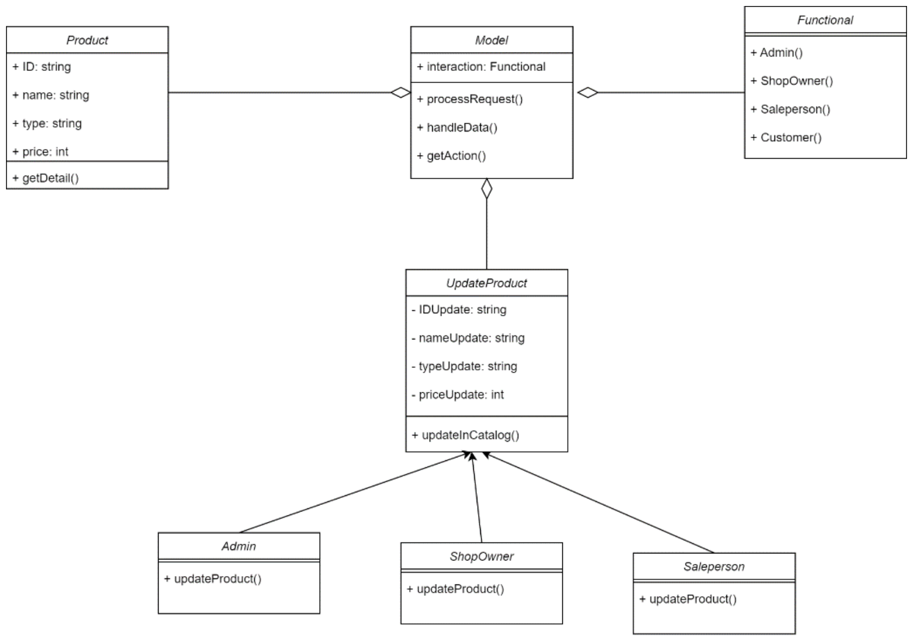
2. **Class diagram for Delete Product**



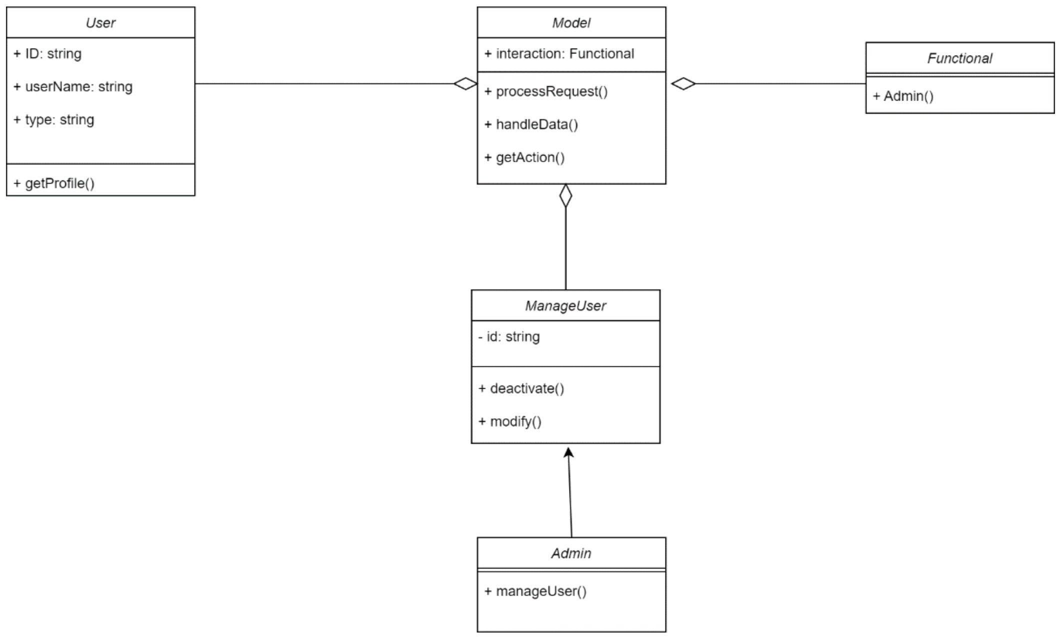
3. Class diagram for Update Product



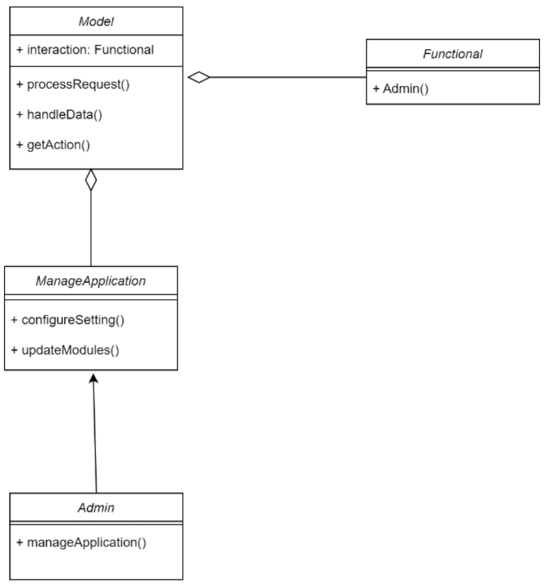
4. Class diagram for Search Product



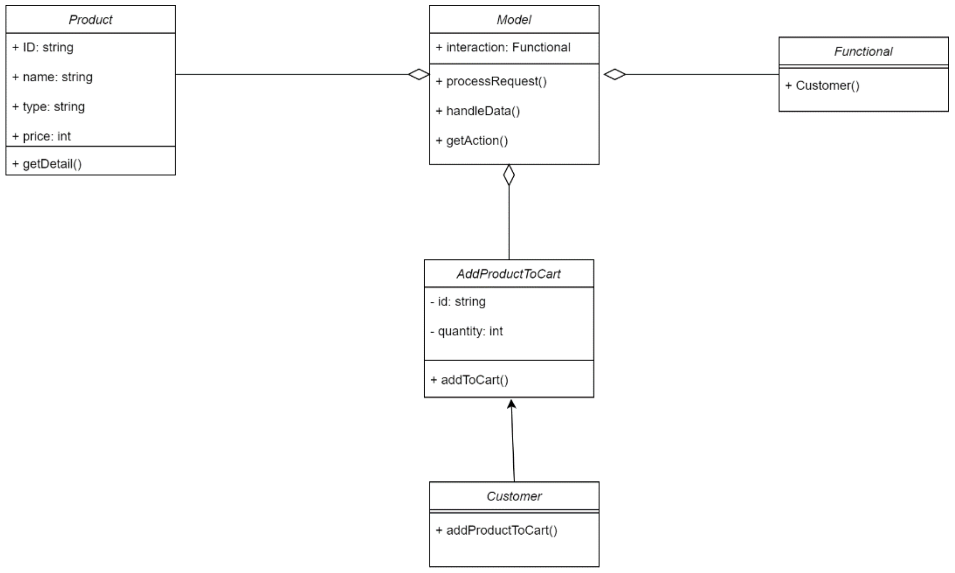
5. Class diagram for Manage User



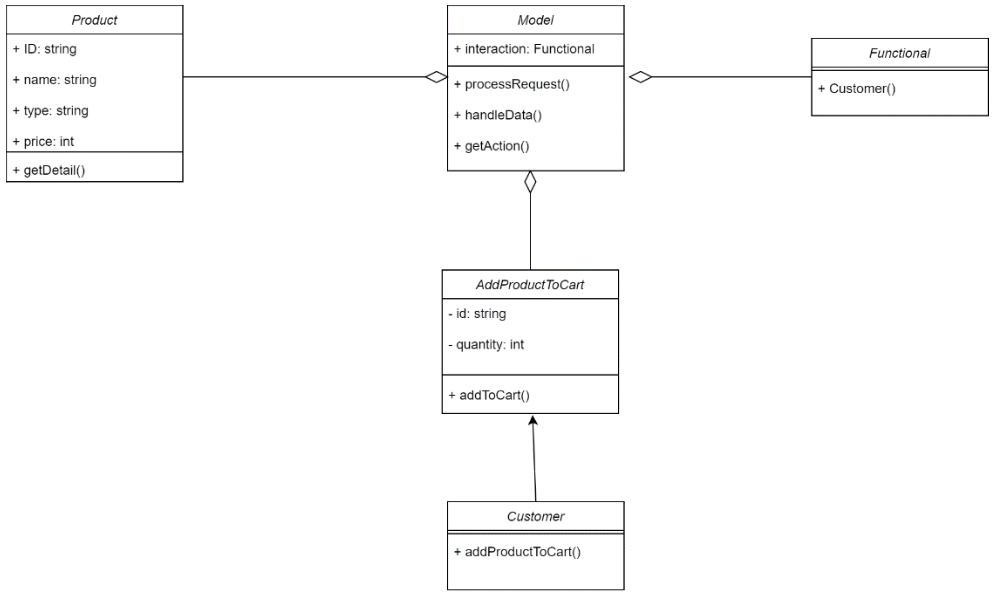
6. Class diagram for Manage Application



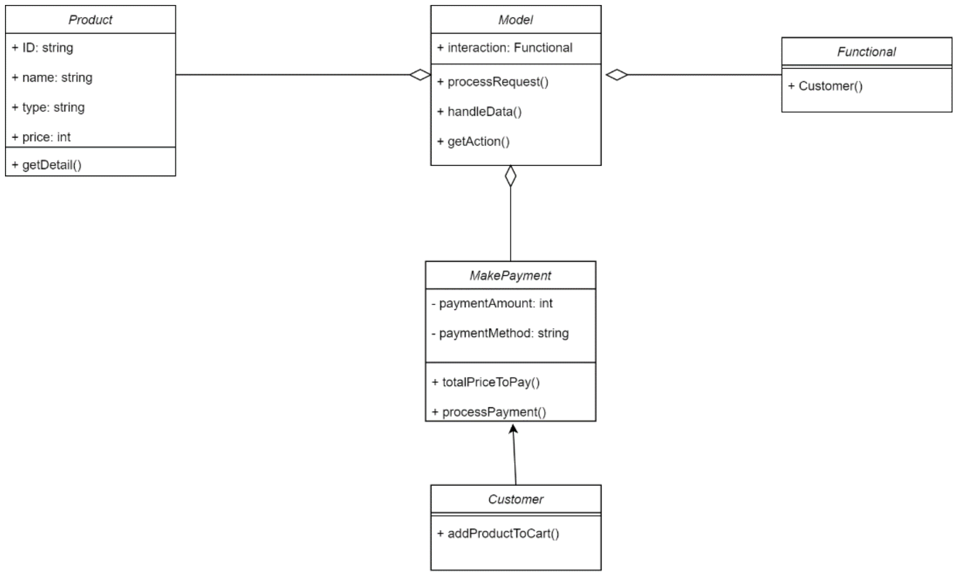
7. Class diagram for Manage Total Revenue



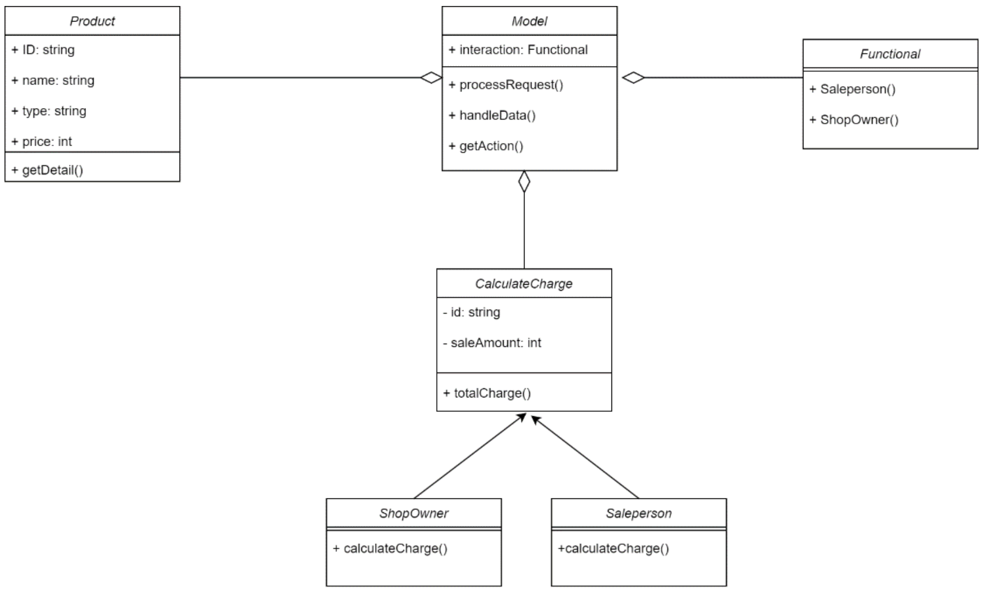
8. Class diagram for Add Product To Cart



9. Class diagram for Make Payment



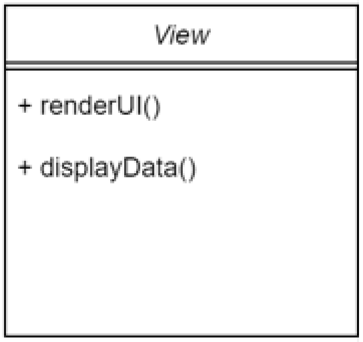
10. Class diagram for Calculate Charge



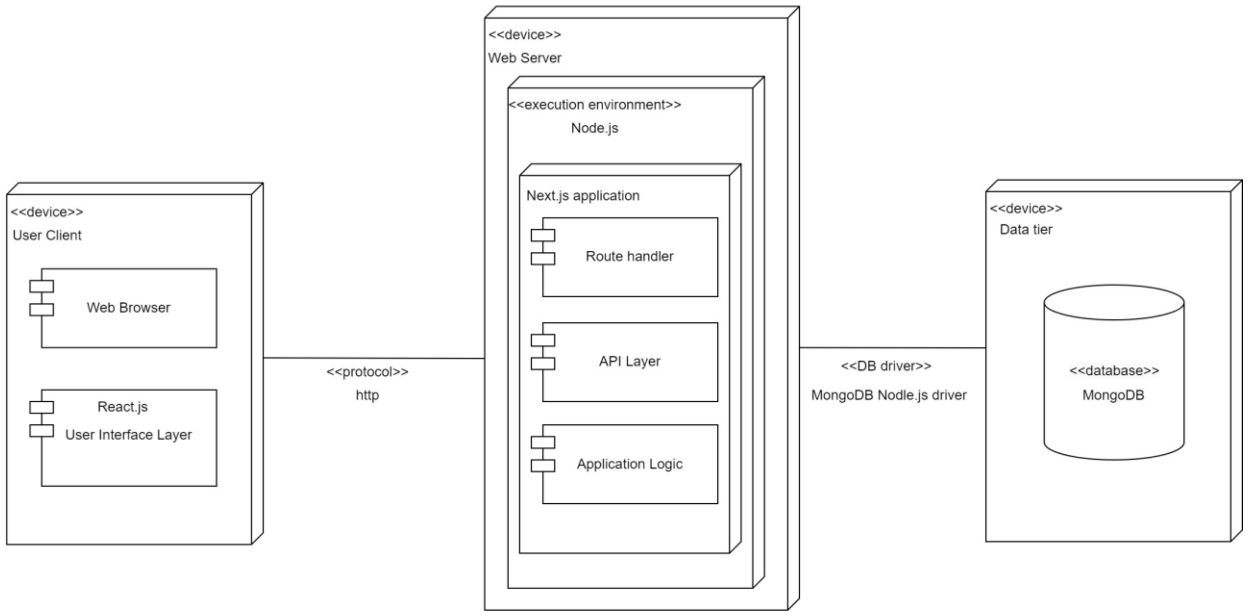
For login, logout, update profile and change password, they are in the general class diagram of Model component.

4.4 Component: View

- **View:** the presentation layer responsible for rendering the user interface (UI) components. In this case, React is used as the front-end library to build the View. It consumes data from the Model and displays it to the users. Because Front-end uses React, the View component would primarily be developed using JSX for rendering UI components and CSS for styling. This class has 2 methods, one takes information provided by the Controller and uses it to render UI, and the other methods are responsible for displaying the rendered UI to the user.



5. Deployment



6. Implementation View

The main folder include:

- Folder **api**:
  - Folder **auth**: [...nextauth].js is a special configuration file in Next.js used by the NextAuth.js library to set up authentication for your application. It allows you to define authentication

providers and customize authentication settings.
    - categories.js: This file is used to handle methods that will be used for processing products, such as "GET", "POST", "PUT", "DELETE",
    - order.js: Used in buying, ordering product,…
    - upload.js: defines a Next.js API route that handles file uploads, utilizing the 'multiparty' library to parse incoming files, the AWS S3 SDK to upload files to a designated S3 bucket, and generating public links for each uploaded file. It requires admin authentication, connects to a MongoDB database, and responds with JSON containing the links to the uploaded files.
    - product.js: Defines a Next.js API route for managing products. It connects to a MongoDB database, requires admin authentication, and handles various HTTP methods.
- Folder **products**:
    - 2 sub folder "delete" and "edit" used to display delete product page and update product page
    - _app.js: wrap the entire Next.js application with a SessionProvider to make authentication session data available to all components.
    - _document.js: customize the basic structure of the HTML document generated by Next.js, including specifying the language and including necessary scripts.
    - categories.js: The main purpose of this code is to create a web application interface for administrators to manage categories and their properties. It allows administrators to perform actions on categories, they are "View", "Create", "Edit", "Delete", "Manage Category Properties", "User Interface", "Confirmation Dialogs"
    - index.js: create a user-friendly homepage for a web application that welcomes the logged-in user with their name and profile image. It uses the next-auth/react package to fetch the user's session information and displays a personalized greeting within the application's layout.
    - orders.js: defines a web application page that retrieves and displays a list of orders. It uses the axios library to fetch order data from an API endpoint, then presents each order's creation date, payment status, recipient's details, and a list of ordered products with quantities in a table format.
    - products.js: defines a web application page that displays a list of products. It uses the axios library to fetch product data from an API endpoint and presents each product's title in a table.

Beside the main folder, we have some supporting folder, include:

- **components**: This folder is used to store React components that can be reused throughout the application. These components can include UI elements, layouts, and other building blocks that make up the user interface. Organizing components in this folder helps maintain a clear structure and promotes reusability.
- **lib**: The "lib" folder usually contains utility functions, helper modules, and other code snippets that provide specific functionalities to the application. These utilities might be used across different parts of the application, making them easy to manage and maintain in a separate folder.
- **models**: In the context of web applications using databases, the "models" folder often contains schema definitions for data models. These models define the structure and attributes of data entities that are stored in the database. They can be used with libraries like Mongoose (for MongoDB) or Sequelize (for SQL databases) to interact with the database.
- **styles**: The "styles" folder typically contains CSS or other styling files used to define the visual appearance of the application. This can include global styles, component-specific styles, and any CSS frameworks or libraries you might be using. Organizing styles in a dedicated folder keeps the styling code separate from the components, enhancing maintainability.