Introduction To Software Engineering

PA3_4

Software Architecture

Version <3.4>

Group 7

21127058	Lưu Đình Huy
21127102	Nguyễn Thành Luân
21127247	Quách Tấn Dũng
21127257	Phạm Trần Minh Duy
21127714	Vũ Phú Trường
21127633	Bùi Tuấn Kiệt

Faculty of Information Technology – HCMC
University of Science

I. Revision History

Date	Version	Description	Author
22/11/2023	1.0	Draw package diagram for front-end.	Lưu Đình Huy
22/11/2023	1.1	Draw package diagram for back-end.	Vũ Phú Trường
22/11/2023	1.2	Draw class diagram for packages: 'main', 'component'.	Quách Tấn Dũng
22/11/2023	1.3	Draw class diagram for packages: 'UI', 'Utils', 'API'.	Nguyễn Thành Luân
22/11/2023	1.4	Draw class diagram for screen's packages: 'component', 'Admin', 'Seller'.	Quách Tấn Dũng
29/11/2023	1.5	Draw class diagrams and write descriptions for package 'main'	Vũ Phú Trường
29/11/2023	1.6	Draw class diagrams and write descriptions for package 'component'	Nguyễn Thành Luân
29/11/2023	1.7	Draw class diagrams and write descriptions for package 'UI'	Quách Tấn Dũng
29/11/2023	1.8	Draw class diagrams and write descriptions for package 'Utils'	Quách Tấn Dũng
29/11/2023	1.9	Draw class diagrams and write descriptions for package 'API' Draw class diagrams and write descriptions for screen's package	Phạm Trần Minh Duy
29/11/2023	2.0	Draw relationships between classes in Front-end	Lưu Đình Huy
29/11/2023	2.1	Draw relationships between classes in Back-end	Bùi Tuấn Kiệt
29/11/2023	2.2	Draw relationships between classes in Front-end	Vũ Phú Trường
29/11/2023	2.3	Draw relationships between classes in Back-end	Bùi Tuấn Kiệt
29/11/2023	2.4	Edit relationships between classes in Back-end	Phạm Trần Minh Duy
8/12/2023	2.5	Edit previous content of Software Architecture Document	Nguyễn Thành Luân

8/12/2023	2.6	Edit previous content of Software Architecture Document	Bùi Tuấn Kiệt
8/12/2023	2.7	Draw Deployment Diagram	Lưu Đình Huy
8/12/2023	2.8	Draw Implementation View Diagram	Vũ Phú Trường
13/12/2023	2.9	Write description for Deployment Diagram	Phạm Trần Minh Duy
13/12/2023	3.0	Write description for Deployment Diagram	Quách Tấn Dũng
13/12/2023	3.1	Write description for Implementation View Diagram	Quách Tấn Dũng
13/12/2023	3.2	Write description for Implementation View Diagram	Phạm Trần Minh Duy
14/12/2023	3.3	Write Report	Lưu Đình Huy
15/12/2023	3.4	Final Review	Lưu Đình Huy

I. Introduction

Purpose

- Requirements Clarification: Use case specifications help clarify and define the functional requirements of the website. By detailing different use cases, you can explicitly state how the system should behave in various scenarios, ensuring that developers have a clear understanding of what is expected.
- Communication: Use case specifications serve as a communication tool between different stakeholders involved in the development process. This includes developers, designers, project managers, and clients. Having a documented reference ensures that everyone is on the same page regarding the system's functionality.
- Scope Definition: Use cases help in defining the scope of the website. By identifying different scenarios and interactions with the system, you can determine the boundaries of what the system is supposed to do, helping to avoid scope creep and ensuring that the project stays focused on its primary objectives.
- Testing: Use cases are valuable for the testing phase of development. Testers can use the documented scenarios to create test cases and ensure that the system behaves as intended under

various conditions. This helps in identifying and resolving potential issues before the website is deployed.

- User Experience (UX) Design: Use case specifications can influence the design of the user interface. Understanding how users will interact with the system in different situations allows designers to create a user-friendly interface that meets the needs of the end-users.
- Risk Management: By detailing different use cases, potential risks and challenges can be identified early in the development process. This allows for proactive risk management, helping to mitigate issues before they become major problems.
- Documentation: Use case specifications serve as documentation for future reference. As the project evolves, and new team members join or changes are made, having a clear and detailed document helps in understanding the original intent and functionality of the system.

Scope

- In Scope:
  - Use-case diagram depicting user's possible interactions with the system.
  - The logical perspective of the system, visualized through Model-View-Controller architecture, and depicted by a package diagram.
  - Each package is depicted by a class diagram of their attributes and functions, what services they provide, how they communicate with others in the system, and their key components.
- Out of Scope:
  - This document does not contain source code, or test case which will be provided in another document.

Definition, Acronyms, Abbreviations

- MVC is a software architecture pattern for creating user interfaces on computers. MVC divides an application into three interoperable parts to separate the way information is processed internally and the way information is presented and received by the user.
- API: An API is a set of rules that allows one software application to interact with another. In the context of a web application, it often refers to the endpoints and methods that enable communication between the client (front end) and server (back end).
- Middleware: Middleware is software that acts as a bridge or intermediary between different applications or components. In web development, middleware is often used to handle tasks such as request processing, authentication, and response formatting.
- Utils: Utilities, or "utils," refer to a set of helper functions or tools that provide commonly used functionalities across different parts of the application. These can include functions for data manipulation, validation, or other generic tasks.
- Class diagram: A class diagram is a type of diagram in UML (Unified Modeling Language) that illustrates the structure and relationships between classes in an application. It typically includes classes, attributes, methods, and associations between classes.

- Package diagram: A package diagram in UML represents the organization and dependencies between different packages or modules in a system. It provides a high-level view of the project's structure, showing how components are grouped into packages.
- Use-case model: A use case model is a representation of the different ways that users can interact with a system. It identifies various use cases (functionalities) and describes how users (actors) interact with the system to achieve specific goals.

References

- (3) PA3 PA4 - YouTube
- Intro2SE-22APCS2-21CLC04-21CLC05-Notes for Project Assignments-Weekly Updated - Google Docs
- https://www.uml-diagrams.org/class-diagrams-overview.html
- https://www.uml-diagrams.org/package-diagrams-overview.html
- https://www.uml-diagrams.org/
- https://www.uml-diagrams.org/uml-25-diagrams.html

Overview

In PA3, our team diligently addressed feedback from PA2, resulting in a refined use-case specification with increased completeness and consistency. Changes made were meticulously logged in the Revision History section. The Software Architecture Document (SAD) was successfully crafted, detailing the system's key components and their organization using MVC, n-tier, and microservices architectural styles. The Logical View section describes each component's responsibilities and connections. Additionally, we developed class diagrams for these components, illustrating classes, attributes, operations, and relationships. The submission adheres strictly to prescribed rules, encompassing well-structured texts, diagrams, and charts.

II. Architecture Goals and Constraints

Constraints

Programming Language

- **Front-end: ReactJS, CSS3, HTML**

  A. HTML (HyperText Markup Language):
  HTML forms the structure of web pages. In an e-commerce site:
  - **Semantic Markup:**
    Use semantic HTML elements to define the structure of content, such as headers, sections, articles, and footers.

- **Page Layout:**
  Structure pages with HTML tags to create the layout of e-commerce site.
- **Forms:**
  Utilize HTML forms for user input, like search bars, login forms, and address details during checkout.

B. CSS3 (Cascading Style Sheets):

CSS is responsible for styling and presentation. In an e-commerce site:

- **Responsive Design:**
  Implement responsive design using media queries to ensure site looks good on various devices and screen sizes.
- **Layout and Flexibility:**
  Use Flexbox and Grid Layout to create flexible and responsive page layouts.
- **Styling Components:**
  Apply styles to HTML elements and React components to create a visually appealing and cohesive user interface.
- **Animations and Transitions:**
  Enhance user experience with CSS animations and transitions for smooth transitions between different states of UI.

C. ReactJS (JavaScript Library):

ReactJS is a JavaScript library for building user interfaces.

- **Component-Based Architecture:**
  Structure UI using reusable components, such as product cards, shopping cart, and navigation bar.
- **State Management:**
  Use React state to manage dynamic data, such as product information, shopping cart items, and user authentication status.
- **Interactivity:**
  Implement interactive features, like adding items to the cart, updating quantities, and handling user interactions, using React event handling.
- **Routing:**
  Use React Router to manage navigation between different pages of e-commerce sites.
- **Fetching Data:**
  Fetch data from a back-end server or API using React's **useEffect** and asynchronous operations.

D. Integration of HTML, CSS3, and ReactJS:

- **JSX Syntax:**
  Use JSX syntax in React components to write HTML-like code, making it easy to integrate HTML and React.
- **Styles in JSX:**
  Apply styles directly in React components using inline styles or use a styling solution like Styled Components or CSS Modules.
- **Component Styling:**
  Style React components based on their states, creating a dynamic and visually appealing user interface.

- # Back-end: NodeJS

A. Server-Side Logic:

Node.js can handle server-side logic, processing business rules, and managing workflows related to e-commerce operations, such as handling product listings, user authentication, and order processing.

B. HTTP Server:

Node.js can be used to create a robust HTTP server to handle incoming requests from clients (web browsers or mobile apps). It efficiently manages the communication between the client and the server.

C. API Development:

Node.js is well-suited for building RESTful APIs. These APIs can be used by the front-end of the e-commerce website to interact with the server for tasks like retrieving product information, updating shopping carts, and processing orders.

D. Real-Time Features:

For e-commerce websites that require real-time features, such as live chat support or real-time inventory updates, Node.js can be used to implement WebSocket functionality for bidirectional communication between clients and the server.

E. Database Connectivity:

Node.js can connect to databases (e.g., MongoDB, MySQL) to store and retrieve data related to products, user information, order history, and more. It is often used with ORMs (Object-Relational Mapping) like Mongoose or Sequelize for database interaction.

F. Middleware Support:

Middleware functions in Node.js can be employed to execute tasks like authentication, authorization, logging, and input validation. These are crucial for securing the e-commerce platform and ensuring data integrity.

G. User Authentication:

Node.js can handle user authentication, allowing users to register, log in, and manage their accounts securely. JSON Web Tokens (JWT) or OAuth can be implemented for secure authentication processes.

H. Payment Gateway Integration:

When integrating with payment gateways for processing transactions, Node.js can handle communication with these external services securely. This includes handling callbacks, confirming successful transactions, and managing payment-related data.

I. Session Management:

Node.js can manage user sessions to keep track of user activities, such as items in the shopping cart, preferences, and order history.

J. Logging and Analytics:

Logs important events and activities within the application for debugging purposes and integrates analytics tools to gather insights into user behavior and platform performance.

K. Error Handling:

Implements robust error handling mechanisms to gracefully manage errors and exceptions, ensuring a smoother user experience and facilitating debugging.

L. Security Measures:

Enforces security measures such as encryption, secure communication (HTTPS), and protection against common web vulnerabilities (e.g., Cross-Site Scripting, SQL Injection).

M. Background Jobs:

Manages background jobs, such as sending asynchronous emails, processing orders, or updating inventory, to improve application responsiveness and efficiency.

N. Scalability:

Designs the architecture with scalability in mind, allowing the system to handle increased loads by adding resources or nodes seamlessly.

O. Content Delivery:

Optimizes content delivery through techniques like content caching, CDN (Content Delivery Network) integration, and efficient handling of static assets.

P. API Security:

Implements security measures for APIs, including rate limiting, input validation, and access controls to protect against potential threats.

Q. Localization and Internationalization:

Supports localization and internationalization, allowing the platform to serve users in different languages and adapt to regional preferences.

R. Version Control:

Utilizes version control systems to track changes, manage collaboration among developers, and facilitate the deployment process.

- # Database: MongoDB

A. Product Storage:

- Store information about products such as name, price, description, images, and stock quantity.
- Support fast search through queries.

B. User Management:

- Store personal information of users such as name, address, email, and encrypted passwords.
- Track the purchase history of each user.

C. Order Processing and Payment:

- Store information about orders, including the purchased products, order value, and payment status.
- Support managing the payment process and tracking the status of orders.

D. Review and Comment Storage:

- Store user reviews and comments about products.
- Allow querying and displaying reviews on the product pages.

E. Management of Promotions and Discounts:

- Store information about promotions, discount codes, and other offers.
- Support discount features for products.

F. Data Objectification:

- Utilize MongoDB's document-oriented data model to represent data flexibly and efficiently.

G. Data Security:

- Implement security measures such as password encryption, user authentication, and access control to ensure data safety.

H. Easy Scalability:

- Leverage MongoDB's support for horizontal scalability, making it easy to scale the database as traffic increases.

Application Environment

- Web hosting: Netlify or Render

Security

- **Data Validation and Sanitization Guardians:** In the dynamic realm of online retail, robust data validation and sanitization protocols stand as the first line of defense. By rigorously validating and cleansing incoming data, the system ensures that only legitimate and sanitized information is processed. This not only thwarts potential injection attacks but also fortifies the overall data integrity, a critical aspect for a secure online shopping experience.
- **Shielding Through Secure Communication:** The cyber landscape demands an unwavering commitment to secure communication channels. In the context of an online marketplace, the implementation of HTTPS (Hypertext Transfer Protocol Secure) becomes imperative. This cryptographic protocol encrypts the data exchanged between the client and the server, ensuring that sensitive information, such as personal and financial details, remains shielded from prying eyes during the transactional journey.
- **Safeguarding Sensitive Customer Information:** In the intricate dance of e-commerce, user data is the crown jewel that demands impeccable protection. Secure storage mechanisms and encryption algorithms are employed to safeguard sensitive customer information. Whether it's personal details, payment credentials, or transaction history, these are cocooned within layers of security, fostering trust and confidence among users.
- **Authentication and Authorization Frameworks:** An e-commerce platform thrives on the trust vested by its users. Authentication and authorization mechanisms play a pivotal role in ensuring that only authorized individuals gain access to critical sections of the system. Multi-factor authentication, robust password policies, and finely tuned access control lists contribute to the creation of a secure environment where user accounts are shielded from unauthorized access.

- **Regular Security Audits and Updates:** The digital landscape is ever-evolving, and so are the tactics of potential threats. Regular security audits, coupled with prompt updates and patches, form an integral part of the proactive approach to cybersecurity. By staying ahead of the curve, the e-commerce platform fortifies itself against emerging vulnerabilities, ensuring a resilient and secure shopping haven for users.
- **Privacy Compliance:** Complying with data protection regulations and privacy standards is not just a legal obligation but a cornerstone of customer trust. Adhering to frameworks like GDPR (General Data Protection Regulation) ensures that user data is handled with utmost care and transparency, fostering a sense of security and reliability among online shoppers.

Performance

- Quick and efficient data retrieval.
- Websites should load quickly and respond to interactions within a reasonable time.
- Website work across different web browsers.
- Website work on various devices.

Implementation Strategy

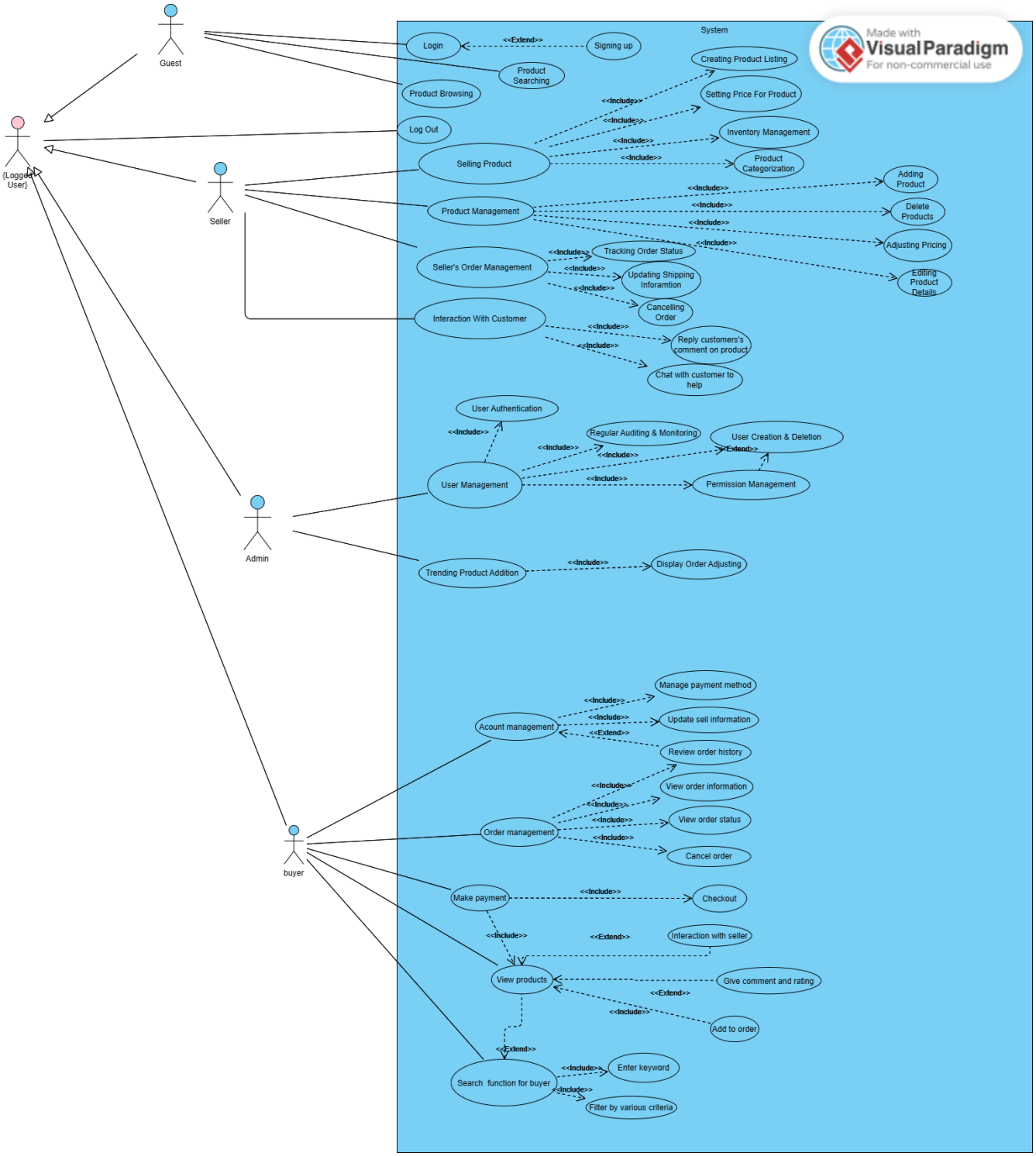
- **ReactJS for Front-End Development:** leveraging ReactJS to build a dynamic and responsive front-end, ensuring an engaging user experience.
- **CSS3 Styling:** Utilizing CSS3 for comprehensive styling, creating visually appealing and user-friendly components.
- **MongoDB NoSQL Database:** Implementing MongoDB as the database solution for its flexibility and scalability in handling diverse and evolving e-commerce data.
- **Deployment on Heroku/Netlify:** Deploying the application on Heroku or Netlify to benefit from their robust hosting environments, ensuring reliability and accessibility.
- **Security Measures:**
    - **Data Validation and Sanitization:** Applying thorough data validation and sanitization practices to mitigate the risk of injection attacks and enhance data integrity.
    - **HTTPS:** Enforcing secure communication between the client and server through HTTPS, safeguarding sensitive information during transit.
- **Secure Cookies:** Implementing secure cookies to protect user data and enhance the overall security posture.
- **Optimizing Performance:**

- **Asset Minimization and Compression:** Minimizing and compressing assets such as images, stylesheets, and scripts to reduce loading times and enhance overall performance.
- **Caching Strategies:** Employing effective caching strategies at various levels to decrease server load and enhance user experience.

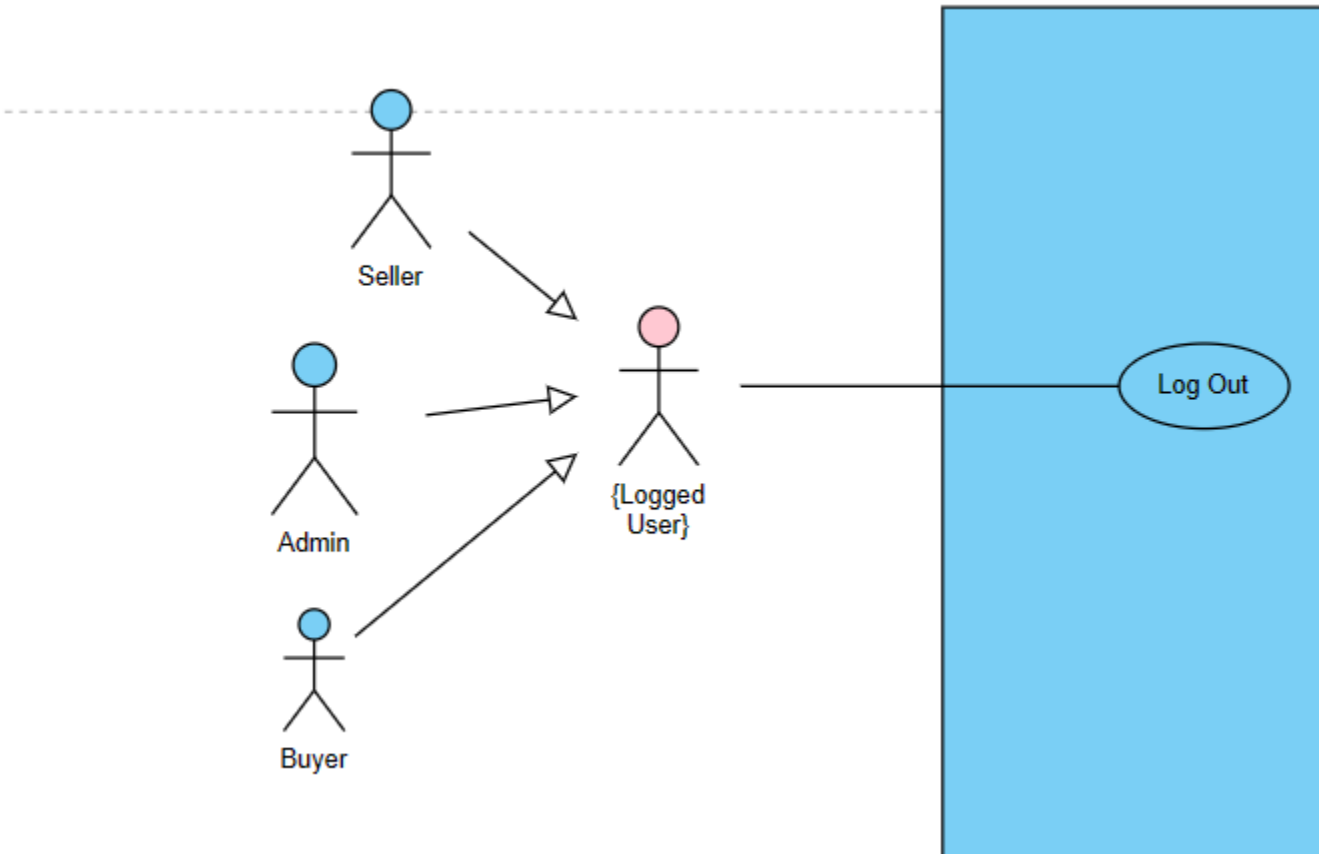
- **Database Optimization:**
  - **Query Optimization:** Streamlining and optimizing database queries to ensure quick and efficient data retrieval, contributing to a seamless user experience.

III. Use-Case Model

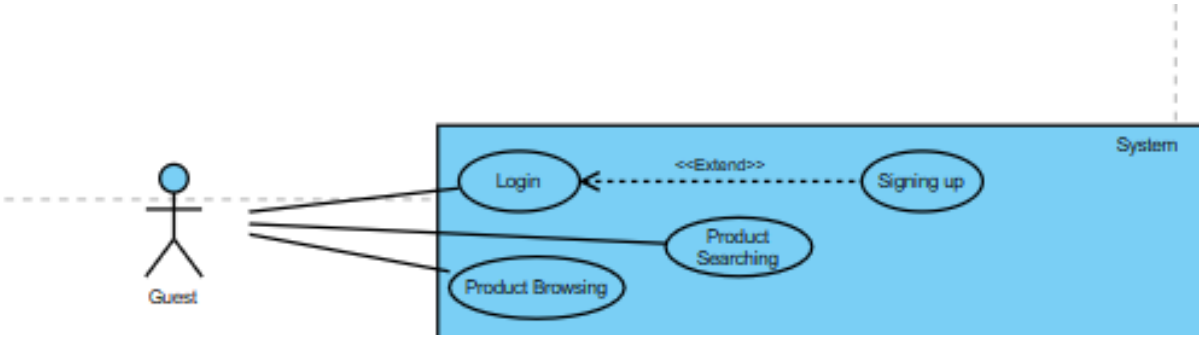
Overview



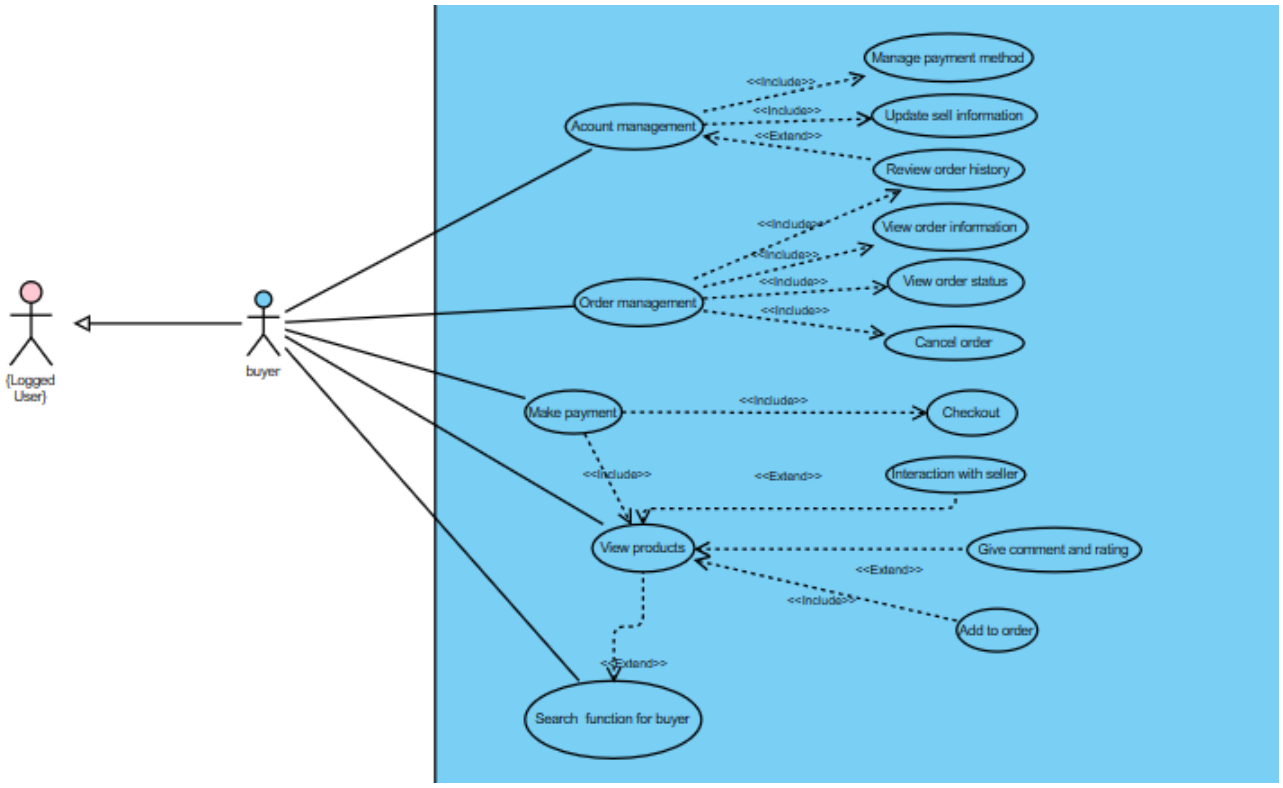
Logged User



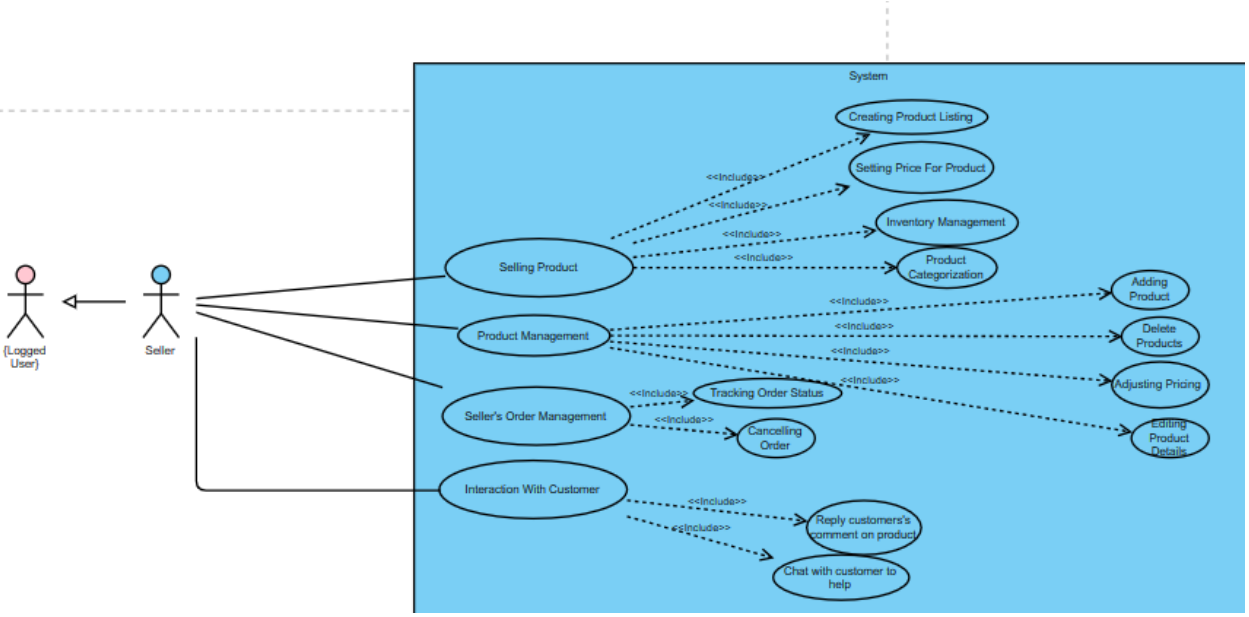
Guest



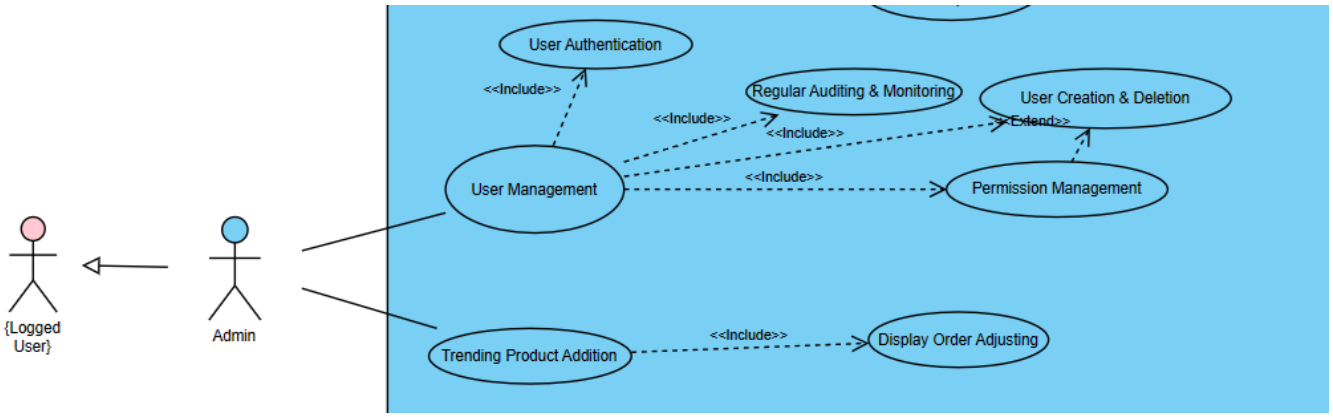
Buyer



Seller

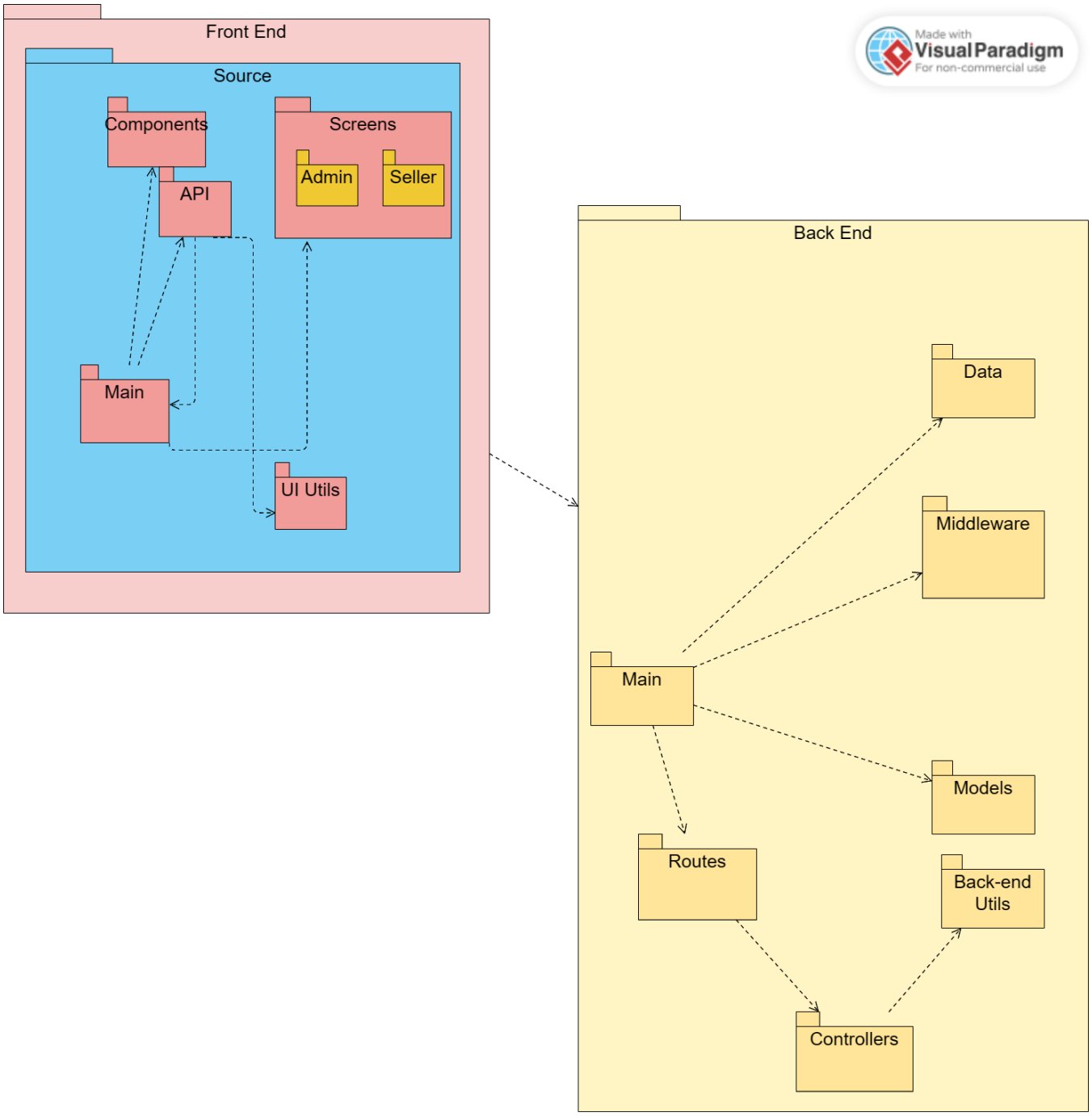


Admin


User Authentication
<<Include>>
Regular Auditing & Monitoring
User Creation & Deletion
<<Include>>
<<Include>>
<<Extend>>
User Management
<<Include>>
Permission Management
{Logged User}
Admin
Trending Product Addition
<<Include>>
Display Order Adjusting

IV. Logical View

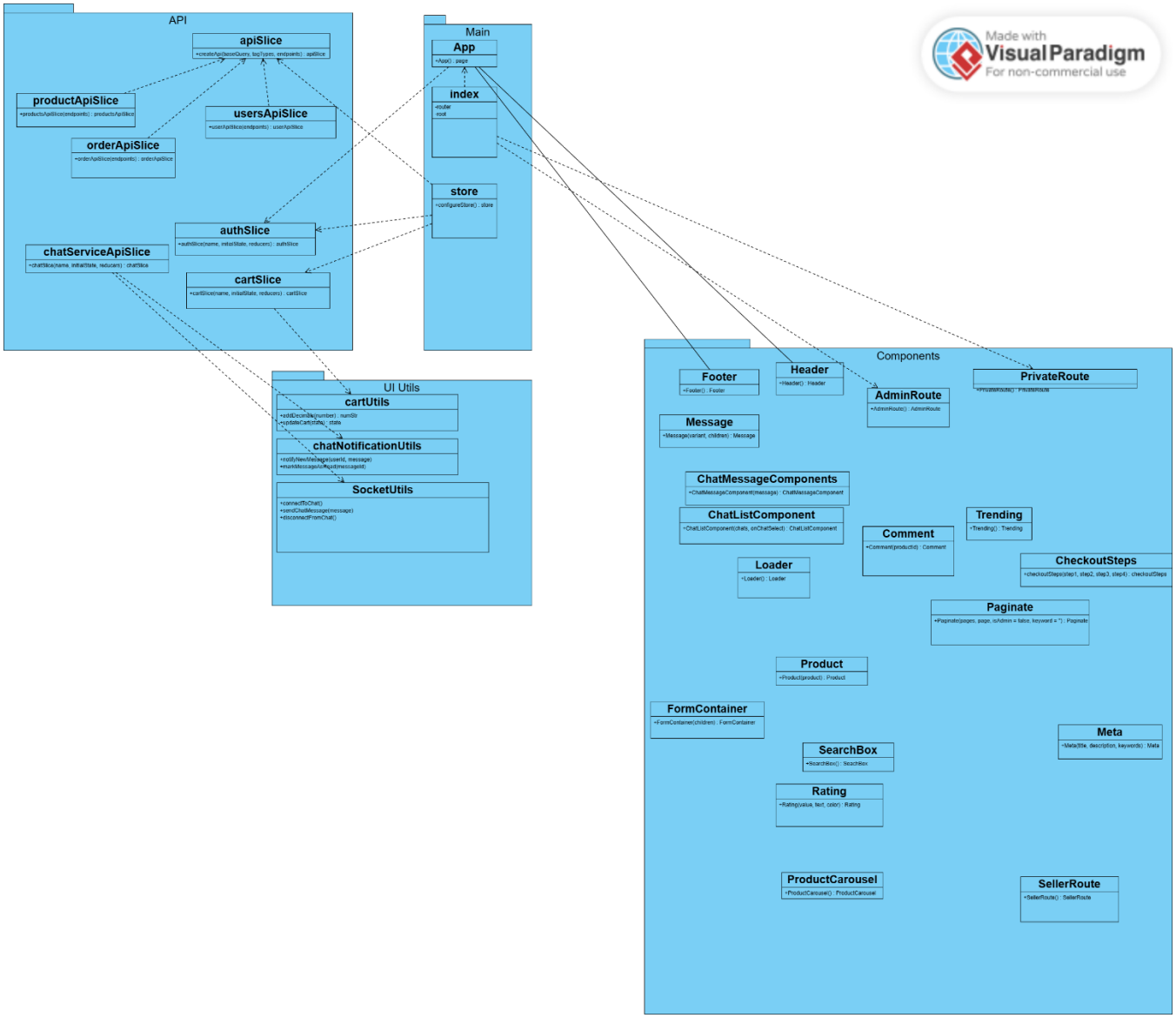
Package Diagram For MVC Model



Components For Each Package

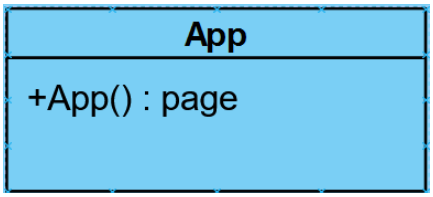
Front-End

Class Diagram of UI And Controller (Main, Api, Components,UI Utils)



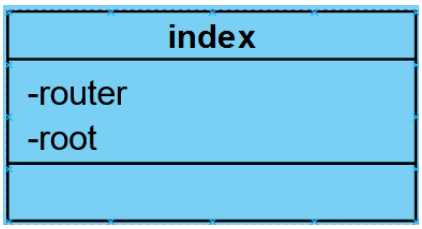
Key classes of package: Main

Class 1: App



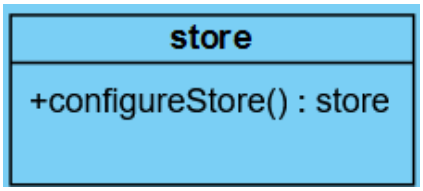
- Description: to set up the basic structure of application, handles user authentication by checking expiration times, and includes components for displaying notifications and managing the layout.
- Methods:
  - App() : page
    - page: basic structure of the app.

Class 2: Index



- Description: to set up the main structure of application, including routing, state management with Redux, and other providers for features like PayPal integration.
- Methods: None

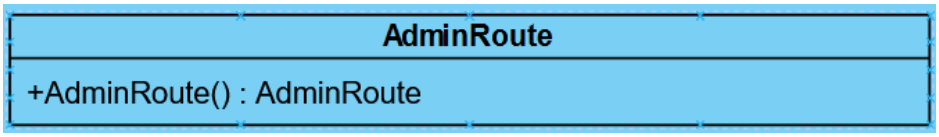
Class 3: store



- Description: to create a Redux store with multiple reducers, including middleware for handling asynchronous API calls. The store is then exported for use in the rest of the application, providing a centralized state management solution.
- Methods:
  - configureStore() : store
    - store: a Redux store configured with specified reducers, middleware, and DevTools settings.

Key classes of package: Component

Class 1: AdminRoute

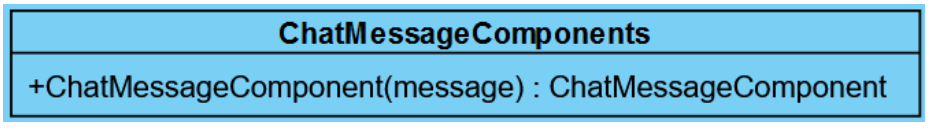


- Description: to protect certain routes that should only be accessible to admin users. It checks the user information stored in the Redux store and conditionally renders the content or redirects the user based on their admin status.
- Methods:
  - AdminRoute() : AdminRoute
    - AdminRoute: a route that should only be accessible to admin users.

Class 2: ChatListComponent

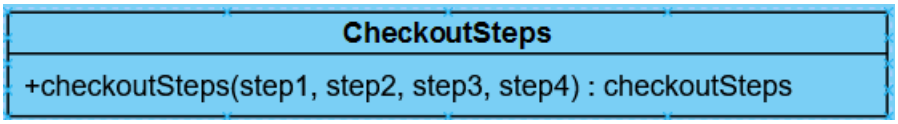
- Description: to provides a visual representation of a list of chat messages (list of different users)
- Methods:
  - ChatListComponent(chats, onChatSelect) : ChatListComponent
    - chats: an array of chat.
    - onChatSelect: a function passing the chat ID that user has clicked on.
    - ChatListComponent: a component

Class 3: ChatMessageComponents



- Description: to provide a visual representation of chat message, based on the author or receiver of the message.
- Methods:
  - ChatMessageComponent(message) : ChatMessageComponent
    - message: an object with ‘author’ and ‘content’.
    - ChatMessageComponent: a component.

Class 4: CheckoutSteps

- 

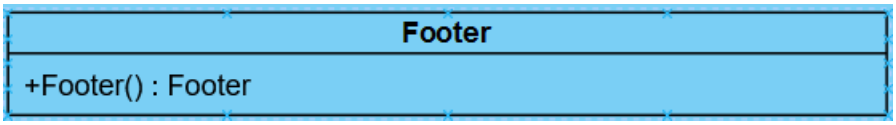
- Description: to provide a visual representation of the steps in the checkout process, and the links are conditionally enabled or disabled based on the completion status of each step.
- Methods:
    - checkoutSteps(step1, step2, step3, step4) : checkoutStep
        - step1: check state: Sign In.
        - step2: check state: Shipping.
        - step3: check state: Payment.
        - step4: check state: Place Order.
        - checkoutStep: a component.

Class 5: Comment

- 

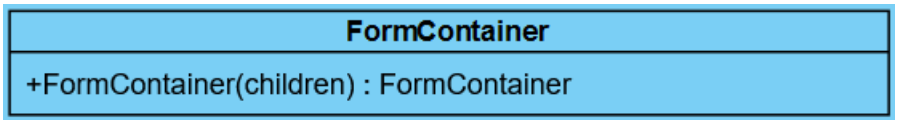
- Description: to provide a form with a text area and a submit button for adding comments to a product.
- Methods:
    - Comment(productId) : Comment
        - productid: an id of a product.
        - Comment: a component.

Class 6: Footer

- 

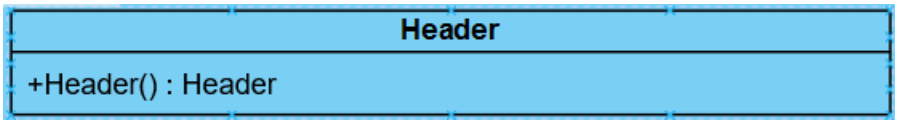
- Description: to provides a simple footer with copyright information, and the year is dynamically updated to the current year.
- Methods:
    - Footer() : Footer
        - Footer: basic structure of the app footer.

Class 7: FormContainer

- 

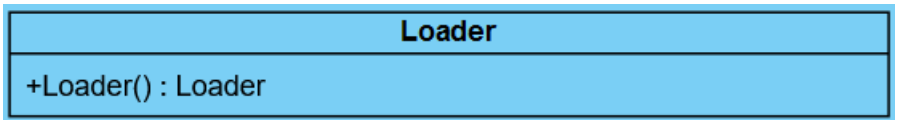
- Description: to provide a standardized layout for forms, ensuring they are centered and responsive. It is flexible in accommodating different form components passed as children.
- Methods:
    - FormContainer(children) : FormContainer
        - children: a component that can wrap and manipulate their content.
        - FormContainer: a component.

Class 8: Header

- 

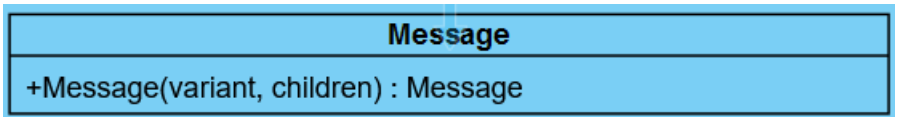
- Description: to provides a responsive navigation bar with links for various user actions, including searching for products, managing the shopping cart, and accessing user and admin functionalities.
- Methods:
    - Header() : Header
        - Header: basic structure of the app header.

Class 9: Loader

- 

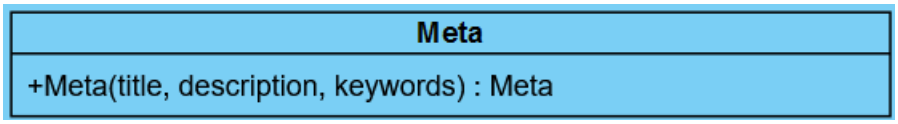
- Description: to provides a simple and customizable loading spinner.
- Methods:
    - Loader() : Loader
        - Loader: basic structure of the app loader symbol.

Class 10: Message

- 

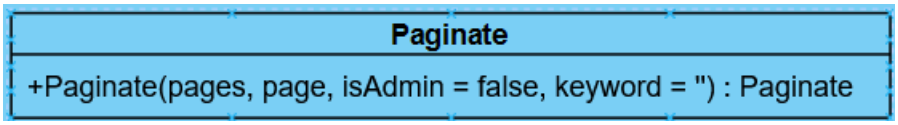
- Description: to provides a simple and reusable way to display alert messages with different styles (variants) like info, success, warning, error, etc.
- Methods:
    - Message(variant, children) : Message
        - variant: a variant of message.
        - children: a content of message.
        - Message: a component.

Class 11: Meta

- 

Description: to provides a convenient way to set page metadata, including title, description, and keywords.
- Methods:
  - Meta(title, description, keywords) : Meta
    - title: the title of the page.
    - description: provides a brief summary or description of the page.
    - keywords: list of keywords relevant to the page's content.
    - Meta: a component.

Class 12: Paginate

- 

- Description: to provides a reusable and conditional pagination control that generates links for navigating between different pages. It considers whether the pagination is for regular users or admin users and adjusts the links accordingly.
- Methods:
  - Paginate(pages, page, isAdmin = false, keyword = '') : Paginate
    - pages: total number of pages.
    - page; current page.
    - isAdmin = false: check if this component is for admin view, default 'false'.
    - keyword = '': a search keyword, default empty string.

Class 13: PrivateRoute

- 

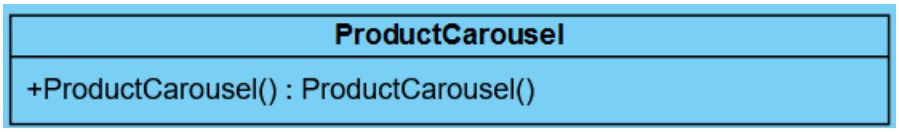
- Description: to provides a mechanism for protecting routes that should only be accessible to authenticated users.
- Methods:
  - PrivateRoute() : PrivateRoute
    - PrivateRoute: a component.

Class 14: Product

- 

- Description: to provides a reusable and styled card for displaying product information, including an image, name, rating, and price.
- Methods:
    - Product(product) : Product
        - product: a product's object.
        - Product: a component.

Class 15: ProductCarousel

- 

- Description: to provide a visually appealing carousel of top products. It fetches the top products, handles loading and error states, and displays the product images along with their names and prices in a carousel format.
- Methods:
    - ProductCarousel() : ProductCarousel
        - ProductCarousel: a component.

Class 16: Rating

- 

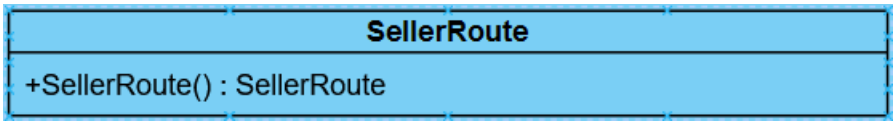
- Description: to provides a visually appealing representation of a star rating, with customizable star colors and optional text.
- Methods:
    - Rating(value, text, color): Rating
        - value: a numerical value of rating.
        - text: additional text of the rating.
        - color: color of the display stars.
        - Rating: a component.

Class 17: SearchBox

- 

- Description: to provides a flexible and controlled search input box and button, allowing users to enter a search keyword and navigate to the corresponding search results page.
- Methods:
    - SearchBox() : SearchBox
        - SearchBox: a component.

Class 18: SellerRoute

- 

- Description: to provide a custom route guard for sellers in the application.
- Methods:
    - SellerRoute() : SellerRoute
        - SellerRoute: a component.

Class 19: Trending

- 

- Description: to fetch and displays a list of trending products.
- Methods:
    - Treding() : Trending
        - Trending: a component.

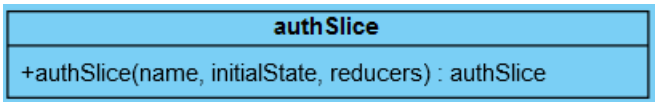
Key classes of package: Api

Class 1: apiSlice

- 

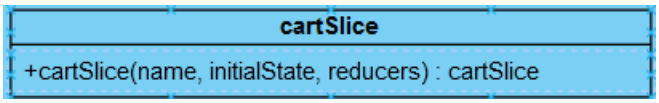
- Methods:
    - createApi(baseQuery, tagTypes, endpoints) : apiSlice
        - baseQuery: a customized base query for handling authentication-related actions
        - tagTypes: tag to differ each type of API.
        - endpoints: an empty object to later define an API endpoint.
        - apiSlice: an API slice.

Class 2: authSlice

- 

- Description: to manage the user's authentication information, updating it when credentials are set and clearing it when the user logs out.
- Methods:
    - authSlice(name, initialState, reducers) : authSlice
        - name: name of slice.
        - initialState: initiate a state for slice.
        - reducers: multiple functions to update user's information and clear local storage when user logout.
        - authSlice: an API slice.

Class 3: cartSlice

- 

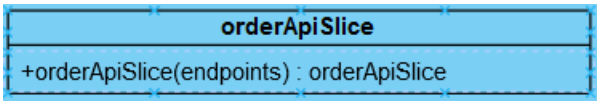
- Description: to manage the state of a shopping cart, allowing items to be added, removed, and the cart to be cleared.
- Methods:
    - cartSlice(name, initialState, reducers) : cartSlice
        - name: name of slice.
        - initialState: initiate a state for slice.
        - reducers: multiple functions to manage states and information of shopping cart.
        - cartSlice: an API slice.

Class 4: chatServiceApiSlice

- 

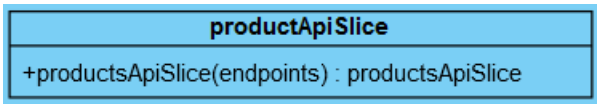
- Description: to manage the state related to a chat application, notifications and handling WebSocket communication.
- Methods:
    - chatSlice(name, initialState, reducers) : chatSlice
        - name: name of slice.
        - initialState: initiate a state for slice.
        - reducers: multiple functions to set up the structure for handling chat-related state, asynchronous operations for sending and receiving messages.
        - chatSlice: an API slice.

Class 5: orderApiSlice

- 

- Description: to provide a clean interface to perform various actions related to order management.
- Methods:
  - orderApiSlice(endpoints) : orderApiSlice
    - endpoints: multiple functions to manage state and information of 'order'.
    - orderApiSlice: an API slice.

Class 6: productApiSlice

- 

- Description: to provide a clean interface for components to perform various actions related to product management.
- Methods:
  - productApiSlice(endpoints) : productApiSlice
    - endpoints: multiple functions to manage state and information of system's products.
    - productApiSlice: an API slice.

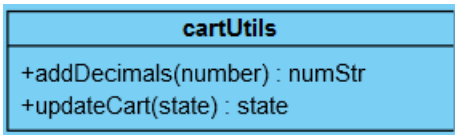
Class 7: usersApiSlice

- 

- Description: to provide a clean interface for components to perform various actions related to user management.
- Methods:
  - userApiSlice(endpoints) : userApiSlice
    - endpoints: multiple functions to manage state and information of system's users.
    - userApiSlice: an API slice.

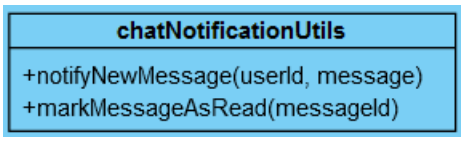
Key classes of package: UI Utils

Class 1: cartUtils

- 

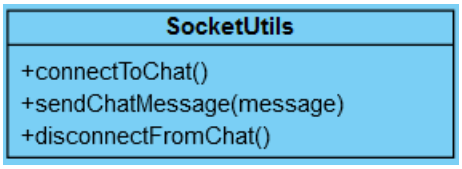
- Description: to calculate items' price, shipping price, tax price, total price based on the cart's content, and updates properties related to cart calculations.
- Methods:
    - addDecimals(number) : numStr
        - to round the decimal number.
        - number: a number.
        - numStr: a string representing the number with two decimal places.
    - updateCart(state) : state
        - to update cart state.
        - state: state of shopping cart before updating.
        - state: state of shopping cart after updating.

Class 2: SocketUtils

- 

- Description: to provide a utility class for managing WebSocket connections related to real-time chat functionality.
- Methods:
    - connectToChat()
        - to establish a WebSocket connection for real-time chat updates.
    - sendChatMessage(message)
        - message: a stringified to JSON message.
    - disconnectFromChat()
        - to close websocket

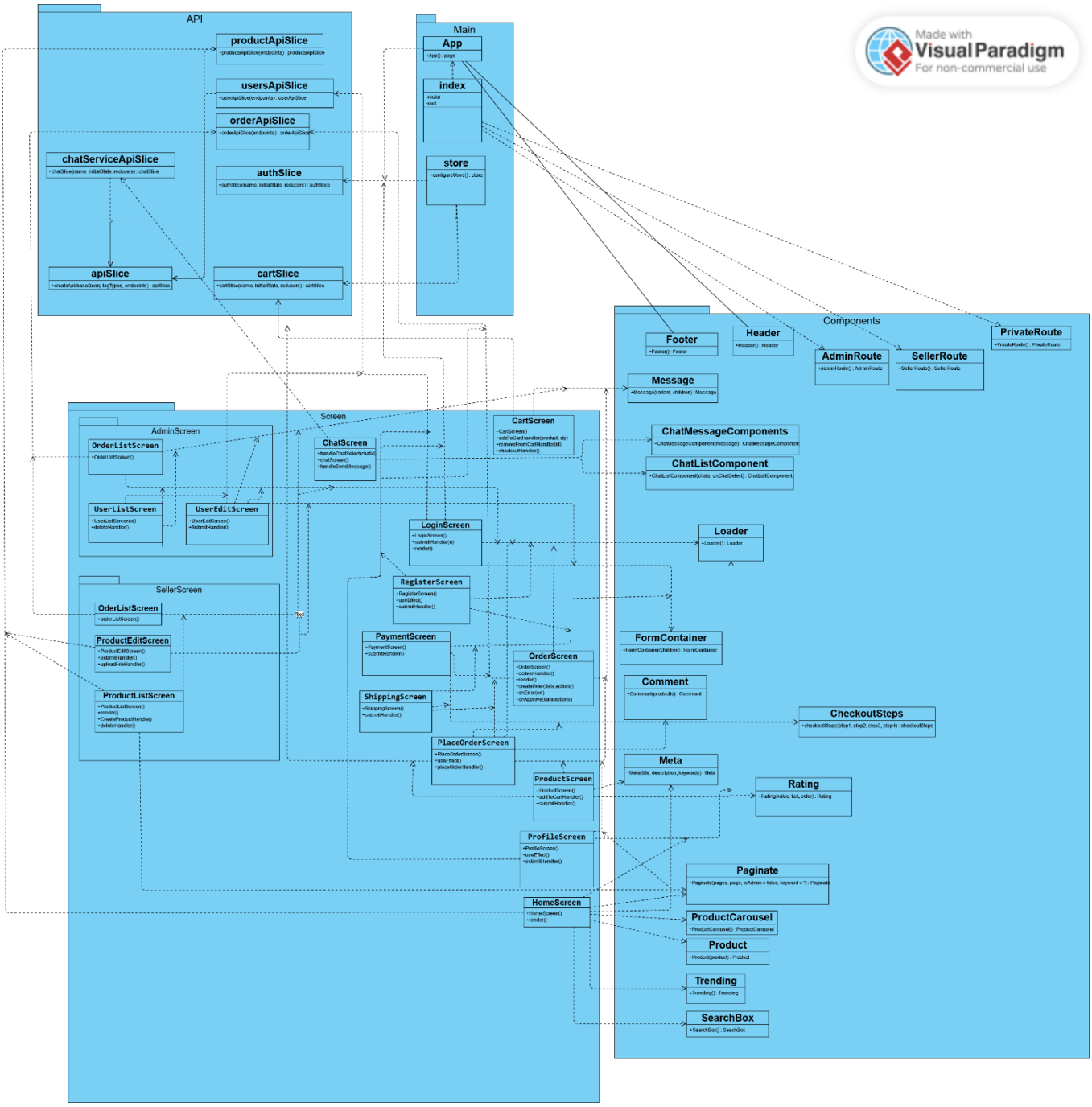
Class 3: chatNotificationUtils

- 

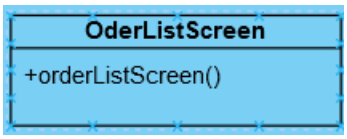
- Description: to provide a utility class for handling chat-related notifications.
- Methods:
    - notifyNewMessage(userId, message)
        - userId: an id of the user that sends the message.
        - message: a notification id.
    - markMessageAsRead(messageId)

- messageId: an id of the message to be marked as read.

Class Diagram of Screen UI: Screens, Main, Components, API



Class 1: OrderListScreen



Description: responsible for rendering the user profile screen. It provides functionality for updating the user's profile information and displays a table of the user's orders.

Method:

1.OrderListScreen():

- show order list screen for user to see order list.

Class 2: userListScreen

Description: The UserListScreen component acts as the main user list screen, providing essential functionalities for user management within the application.

Method:

1.UserListScreen(id):

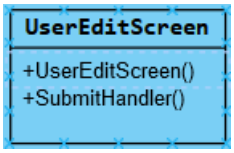
- User Fetching: Utilizes the useGetUsersQuery hook to fetch and display a list of users.
- Loading Handling: Displays a loading spinner during user data fetching for a better user experience.
- Error Management: Handles errors during user fetching and shows an error message if the operation fails.
- User Table Rendering: Renders a table with user information, including ID, name, email, admin status, and interactive actions.
- Interactive Elements: Provides interactive elements such as edit and delete buttons for user account management.

2.deleteHandler():

- Confirmation Prompt: Utilizes window.confirm to display a confirmation prompt for user deletion.
- Delete User Mutation: Initiates the deletion of the user with the specified ID using the deleteUser mutation.
- User List Update: Triggers a list update through the refetch function after a successful user deletion.

- Error Handling: Displays error messages using the toast library in case of deletion failure to inform users effectively.

Class 3: userEditScreen



Description: provide administrators with the capability to edit user details, including name, email, and administrative privileges, through a user interface. Specific functionalities include displaying an editing interface, interacting with an API using mutations for updating user information, handling various states such as loading and errors, redirecting users upon successful updates, and providing success and error notifications.

Method:

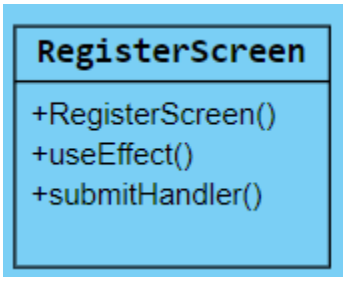
1.userEditScreen()

- Display the user information editing interface.
- Show a loading indicator while fetching or updating data.
- Display an error message if there is an error during data fetching or updating.
- Populate form fields with the retrieved user information.
- Trigger the submitHandler when the user clicks the "Update" button.
- Redirect the user to the user list after a successful update.

2. submitHandler()

- Prevent the default behavior of the form.
- Invoke a mutation to update the user information.
- Display a success message using react-toastify.
- Refresh the user information.
- Redirect the user to the user list after a successful update.
- Display an error message if there is an error during the update process.

Class 4: RegisterScreen



Description: input their information, including personal details and login credentials, through a visually appealing form. The component ensures a seamless registration experience, performing validation checks on the provided information. Upon successful registration, users are smoothly directed to their intended destination within the application. The interface maintains a clean and responsive design, offering an intuitive experience for users new to the platform.

1.RegisterScreen:

- Manages the overall user registration process.
- Orchestrates various interactions and behaviors for a cohesive user experience.

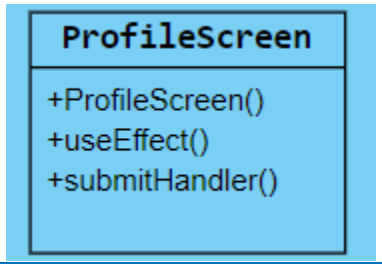
2.useEffect:

- Controls navigation flow based on specific conditions.
- Ensures a seamless transition for the user without referencing any specific function or variable names.

3.submitHandler:

- Handles form submissions during the registration process.
- Ensures a smooth and secure user registration experience without relying on specific functions or variable identifiers.

Class 5: ProfileScreen



Description: manages user profiles in a React application, offering a seamless experience for updating personal information and viewing order details. Users can effortlessly modify their profile information, such as name, email, and password, through a well-designed form. The component also provides a clear overview of the user's order history, displaying essential details such as order ID, date, total amount, payment status, and delivery status in a structured table. The design ensures responsiveness and a user-friendly interface, enhancing the overall user experience for profile management and order tracking.

Method:

1.ProfileScreen:

- Manages user profiles, allowing users to update personal details and view order history.
- Ensures a smooth and responsive user interface for an enhanced user experience.

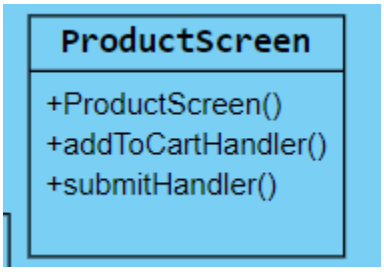
2.useEffect:

- Controls specific actions based on certain conditions or changes within the component.
- Enhances the user experience by handling asynchronous operations and interactions.

3.submitHandler:

- Handles the submission of user profile updates, ensuring a secure and seamless process.
- Implements validation checks to maintain data integrity and provide feedback to users.

Class 6: ProductScreen



Descriptions: showcases and manages product details in a React application, enabling users to explore product information, add items to their cart, and submit reviews. The interface is designed for a seamless

user experience, providing clear visuals of the product alongside essential details. Users can effortlessly navigate through reviews, submit their own feedback, and interact with the product through the intuitive user interface. The component ensures a responsive design, making it easy for users to engage with product-related actions, enhancing the overall shopping experience.

Method:

1.ProductScreen:

- Manages the display and interaction for individual product details.
- Allows users to explore product information, view reviews, and submit their own feedback.
- Implements a responsive design for an enhanced user experience.

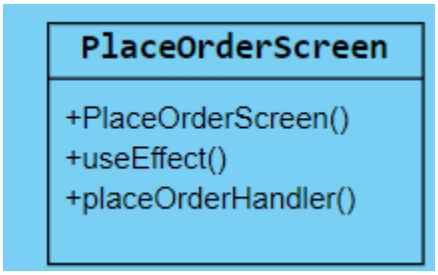
2.addToCartHandler:

- Handles the process of adding the displayed product to the user's shopping cart.
- Initiates the appropriate actions to update the cart state.

3.submitHandler:

- Manages the submission process for user reviews on the displayed product.
- Performs necessary actions to send the review data to the backend for processing.
- Updates the displayed product details based on the submitted review.

Class 7: PlaceOrderScreenn



Descriptions: facilitates the placement of an order in a React application, guiding users through the final steps of the checkout process. Users can review and confirm their shipping address, selected payment method, and a summary of ordered items. The interface presents a clear breakdown of

costs, including item prices, shipping fees, taxes, and the total order amount. The "Place Order" button initiates the order creation process, updating the backend with the necessary information. In case of success, users are redirected to an order confirmation page. The component ensures a seamless user experience, maintaining responsiveness and clarity in conveying order details.

Method:

1.Main Component (PlaceOrderScreen):

- Manages the final steps of the checkout process for placing an order.
- Displays shipping details, payment method, and a summary of ordered items.
- Allows users to review the order details before confirming the purchase.

2.useEffect:

- Monitors the cart state and redirects users to appropriate steps if necessary.
- Ensures users proceed through the checkout process in the correct sequence.

3.placeOrderHandler:

- Handles the user's action to place an order.
- Initiates the order creation process by sending relevant information to the backend.
- Redirects the user to an order confirmation page upon successful order placement.

Class 8: ShippingScreen

Descriptions: manages the user's shipping information during the checkout process. It provides a form where users can input their shipping details, including address, city, postal code, and country. The entered information is then dispatched to the Redux store, updating the application state. Upon submission, users are directed to the next step in the checkout process, contributing to a seamless and user-friendly experience. The

component also integrates with a set of checkout steps for visual guidance through the process.

Method:

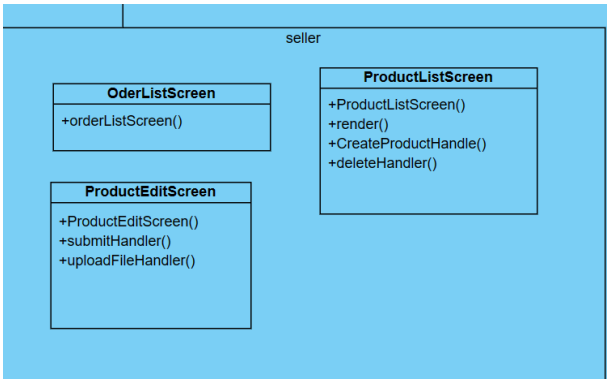
1.ShippingScreen:

- Renders a form containing input fields for the shipping address, city, postal code, and country.
- Utilizes local state variables to manage user input.
- Displays a "Continue" button to submit the form.

2.Submit Handler Function:

- Prevents the default form submission behavior.
- Dispatches the shipping address details to the Redux store.
- Navigates users to the payment step.

Keyclass of package: Seller



Class 1: OrderListScreen

Description: responsible for rendering the user profile screen. It provides functionality for updating the user's profile information and displays a table of the user's orders.

Method:

1.useEffect():

- Auto-population: Automatically fills the form fields with the user's existing name and email when the component loads.
- Dynamic Updates: Listens for changes in specific user data and triggers a re-render to ensure the displayed information remains current.

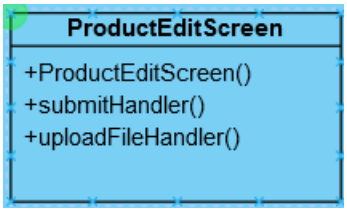
2.submitHandler():

- Form Submission: Manages the form submission event when the user updates their profile.
- Password Validation: Checks if the entered passwords match and provides user feedback if they don't.
- Profile Update: Utilizes a mutation hook to send a request for updating the user's profile on the server.
- Redux Dispatch: Updates the local Redux store with the modified user information upon a successful profile update.
- User Feedback: Displays concise success or error messages using an external library for a streamlined user experience.

3. ProfileScreen():

- Form Rendering: Renders a user-friendly form for updating profile information, including fields for name, email, password, and confirm password.
- State Management: Utilizes controlled components to manage form state, ensuring a seamless and responsive user interface.
- Loader Display: Shows a loading indicator during the profile update process, providing visual feedback to the user.
- Styling: Applies Bootstrap components for consistent and visually appealing styling.
- Order History: Fetches and displays the user's order history in a table format using a query hook.

Class 2: ProductEditScreen



Description: Responsible for rendering the screen for editing a product. It manages the form for updating product details, including name, price, image, brand, category, count in stock, and description. The component provides user feedback through loading indicators, success messages, and error messages. Additionally, it allows the user to navigate back to the product list.

Methods:

1. submitHandler():

- Form Submission: Manages the form submission event when the user updates a product.

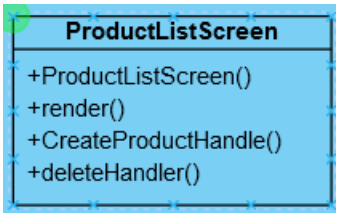
- Product Update: Utilizes a mutation hook to send a request for updating the product on the server.
- Redux Dispatch: Updates the local Redux store with the modified product information upon a successful update.
- User Feedback: Displays concise success or error messages using the 'react-toastify' library for a streamlined user experience.

2. uploadFileHandler():

- File Upload: Manages the event for uploading a product image.
- FormData Handling: Creates a FormData object and appends the selected file before making a request to update the product image.
- User Feedback: Displays success or error messages using the 'react-toastify' library.

3. ProductEditScreen() : initiate the product edit screen

Class 3: ProductListScreen



Description: The ProductListScreen component is responsible for rendering a screen that displays a list of products. It manages the retrieval of product data, allows users to create new products, and provides functionality for deleting existing products. The component features Bootstrap styling for a visually appealing layout.

Methods:

1. deleteHandler():

- Confirmation Dialog: Prompts the user with a confirmation dialog before proceeding with the product deletion.
- Deletion: Deletes the selected product using a mutation hook and triggers a refetch to update the displayed product list.
- User Feedback: Displays error messages using the 'react-toastify' library in case of deletion failure.

2. createProductHandler():

- Confirmation Dialog: Prompts the user with a confirmation dialog before proceeding with the creation of a new product.

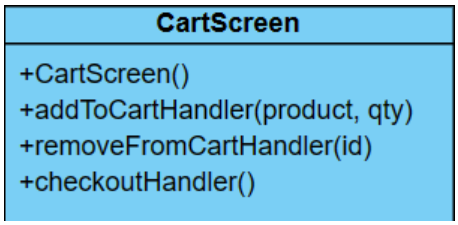
- Product Creation: Uses a mutation hook to create a new product and triggers a refetch to update the displayed product list.
- User Feedback: Displays error messages using the 'react-toastify' library in case of creation failure.

3. Render():

- Page Header: Renders a header with the title "Products."
- Create Product Button: Displays a button that, when clicked, triggers the createProductHandler() method to create a new product.
- Loading Indicators: Displays loading indicators during the creation and deletion processes.
- Data Fetching and Error Handling: Utilizes the useGetProductsQuery hook to fetch product data, displaying a loader during the loading phase and an error message if an error occurs.
- Table Rendering: Renders a table with product details, including ID, name, price, category, and brand.
- Edit and Delete Buttons: Provides buttons for editing and deleting each product, with the edit button linking to the product editing screen and the delete button triggering the deleteHandler() method.
- Pagination: Utilizes the Paginate component to render pagination controls for navigating through product pages.

4. ProductListScreen() : initiate the product list screen

class 4: CartScreen



Description: The CartScreen component is responsible for rendering the shopping cart, allowing users to view and manage items in their cart. It provides functionality for adding, removing items, and proceeding to checkout. The component utilizes Redux for state management and Bootstrap components for styling.

Methods:

1. addToCartHandler():

- Adding to Cart: Dispatches the addToCart action to add a product to the cart.
- Quantity Selection: Allows users to select the quantity of items using a FormControl dropdown.

2. removeFromCartHandler():

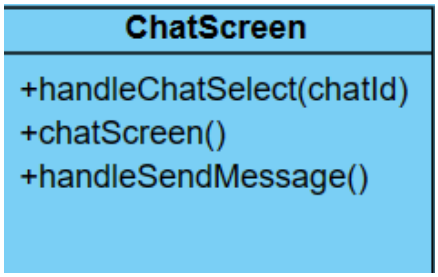
- Removing from Cart: Dispatches the removeFromCart action to remove a product from the cart.

3. checkoutHandler():

- Redirect to Login: Navigates the user to the login screen with a redirect to the shipping screen upon checkout.

4. CartScreen() : initiate the cart screen

class 5: ChatScreen



Description: The ChatScreen component manages a chat room, allowing users to send and receive messages. It utilizes Redux for state management and includes components for displaying a list of chats, individual chat messages, and a text input for composing and sending messages.

Methods:

1. handleChatSelect():

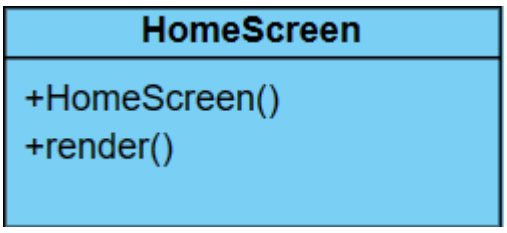
- Chat Selection: Logs the selected chat ID when a chat is selected.
- Future Implementation: Placeholder for setting the selected chat and retrieving messages for that chat.

2. handleSendMessage():

- Message Sending: Dispatches the sendMessage action to send a new message.
- Input Reset: Clears the input field after sending a message.

3. ChatScreen() : initiate the chat screen

Class 6: HomeScreen



Description: The HomeScreen component is responsible for displaying a list of products on the home page. It utilizes the useGetProductsQuery hook from the productsApiSlice to fetch and display the products. The component also includes features such as product carousels, pagination, and error handling.

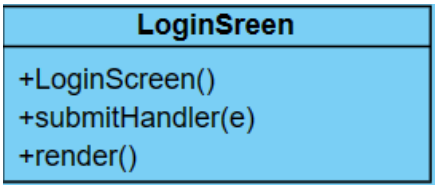
Methods:

1. Render():

- Product Carousel or Go Back Button:
  - Renders a ProductCarousel if there is no search keyword (!keyword).
  - Otherwise, renders a 'Go Back' button linking to the home page.
- Loading, Error, or Product Display:
  - Displays a loader if data is still loading.
  - Displays an error message if there is an error fetching data.
  - Otherwise, maps through the data.products array and renders individual Product components.
- Pagination:
  - Renders a Paginate component to handle pagination, passing down the number of pages, current page, and the search keyword.
- Meta Component:
  - Renders a Meta component, which likely handles setting meta tags for SEO purposes.

2. HomeScreen() : initiate home screen

class 7: LoginScreen



Description: The LoginScreen component is responsible for rendering a sign-in form, allowing users to log in to the application. It utilizes the useLoginMutation hook from the usersApiSlice for handling user authentication. The component also includes features such as redirecting after successful login and handling loading states.

Methods:

1. submitHandler():

- Form Submission: Handles the form submission event.
- User Authentication: Uses the useLoginMutation hook to send a request for user authentication.

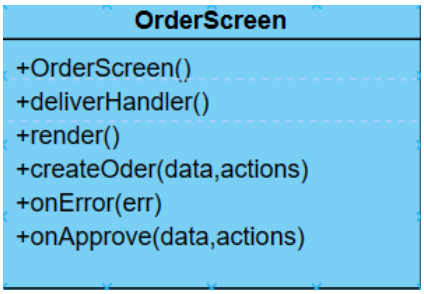
- Dispatch Credentials: Dispatches the setCredentials action to store user information in the Redux store upon successful authentication.
- Redirect: Navigates to the specified redirect location after successful login.
- Error Handling: Displays error messages using the 'react-toastify' library if authentication fails.

2. Render():

- FormContainer: Utilizes a FormContainer component for styling, providing a centered and formatted layout.
- Form: Renders a form with input fields for email and password.
- Disables the submit button and displays a loader during the authentication process.
- Registration Link: Includes a link to the registration page for new customers.

3. LoginScreen() : initiate login screen

class 8: OrderScreen



Description: The OrderScreen component is responsible for displaying detailed information about a specific order. It provides functionality to view shipping details, payment method, order items, and order summary. Additionally, it allows administrators to mark orders as delivered and handles the payment process through PayPal integration.

Methods:

1. onApprove():

- Order Capture: Captures the order on PayPal approval and updates the order status.
- Success Notification: Displays a success notification when the order is paid.

2. onError():

- Error Handling: Displays an error message in case of payment errors.
- createOrder():
- Order Creation: Creates an order with PayPal, specifying purchase units and total amount.

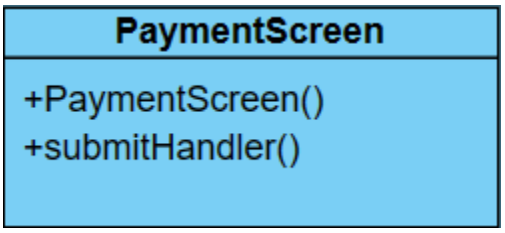
3. deliverHandler():

- Order Delivery: Initiates the process of marking an order as delivered by calling the deliverOrder mutation.
- Refetch: Updates the order details after marking the order as delivered.

4. Render():

- Loading/ Error States:
- Displays a loader during data fetching and shows an error message if there is an issue.
- Order Details:
- Displays shipping information, payment method, and a list of ordered items.
- Order Summary:
- Provides a summary of items, shipping, tax, and total cost.
- Payment Handling:
- Renders PayPal buttons for payment.
- Deliver Button:
- Displays a "Mark as Delivered" button for administrators to update the order status.

class 9: PaymentScreen



Description: The PaymentScreen component handles the selection of payment methods during the checkout process. It ensures that users have chosen a shipping address before proceeding to payment. The primary payment method supported is PayPal, and the selected method is saved in the Redux store for future reference.

Methods:

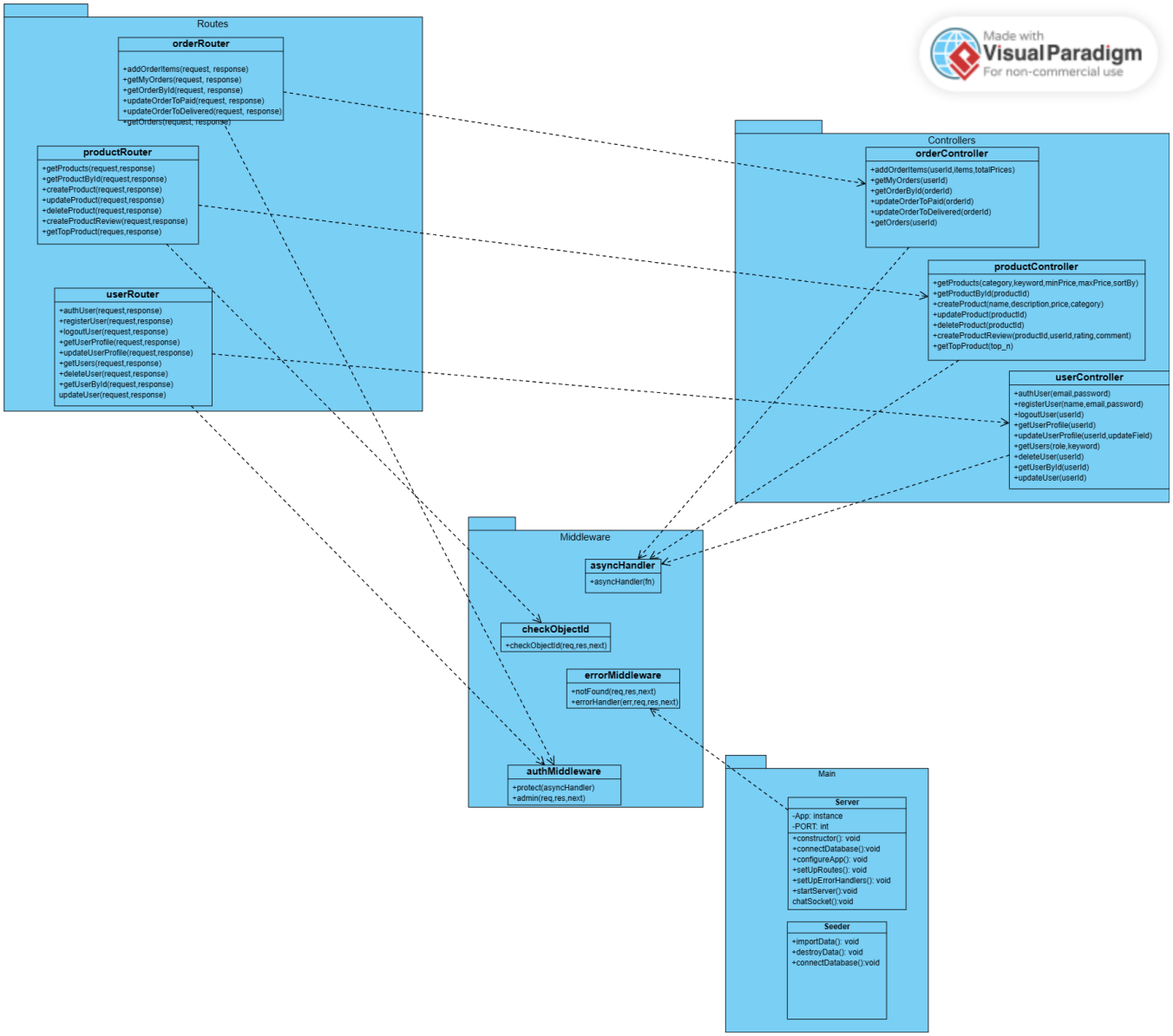
1. submitHandler():

- Form Submission: Handles the submission of the payment method form.
- Save Payment Method: Dispatches an action to save the selected payment method to the Redux store.
- Navigate to Place Order: Redirects the user to the "Place Order" screen after successfully saving the payment method.

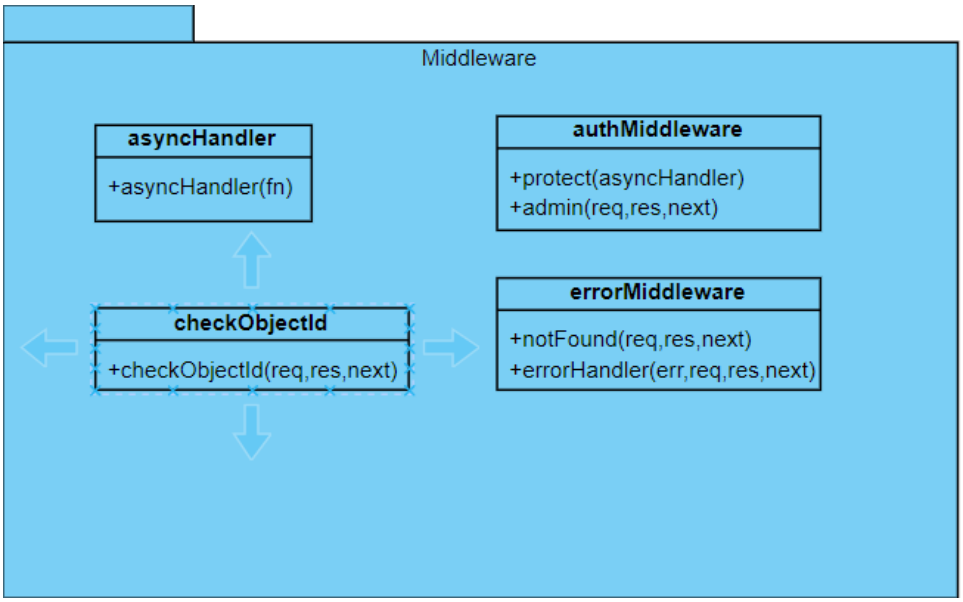
2. PaymentScreen() : initiate payment screen

Back-End

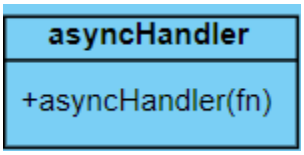
Class Diagram Of Back-End Service: Routes, Controllers, Middleware, Main



Key classes of package Middleware



Class 1: asyncHandler



Description: The asyncHandler function is a higher-order function that takes another function fn as its input parameter. This function is designed to wrap asynchronous route handlers in Express.js to handle promise rejections

Methods:

1.asynHandler(fn)

- Parameters: fn This is the asynchronous route handler function that you want to wrap. It's assumed to be a function that takes three parameters: req (request), res (response), and next (next middleware function).
- Return Value: The asyncHandler function returns a new function that takes the same three parameters: req, res, and next. This new function wraps the execution of the original fn function.
- Promise Handling: Inside the returned function, Promise.resolve() is used to create a resolved promise with the result of calling the original fn function with req, res, and next. This allows the fn function to be asynchronous and return a promise.

Class 2: **authMiddleware**

Description: This code defines middleware functions for user authentication and authorization in a Node.js application using Express. The code uses JSON Web Tokens (JWTs) for authentication and checks if a user is authorized to access certain routes.

Method:

1. **protect(asyncHandler):**
   - **Async Function Signature:** asyncHandler is a custom middleware that simplifies error handling for asynchronous function
   - **The protect function** ensures that routes it protects can only be accessed by clients with a valid and verified JWT, and it attaches the corresponding user information to the request for further processing by downstream middleware or routes. If the token is missing or invalid, it returns a 401 status with an appropriate error message.
2. **admin(req,res,next)**
   - **Middleware Function Signature:** The admin function takes three parameters: req (request), res (response), and next (next middleware in the stack).
   - **The admin function** acts as middleware to restrict access to routes or resources that require administrative privileges. It checks whether the user making the request has the isAdmin property set to true. If the check passes, the request is allowed to proceed; otherwise, a 401 status is returned, indicating unauthorized access as a non-administrator. This middleware is typically used in conjunction with the protect middleware to ensure that a user is both authenticated and has the necessary administrative permissions to access certain parts of the application.

Class 3: checkObjectId(req,res,next)

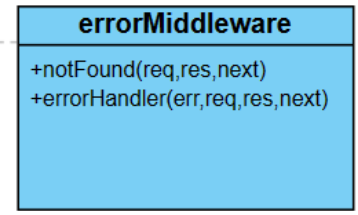
Description: The checkObjectId function is an Express middleware designed to validate whether a parameter in the request (req.params.id) is a valid MongoDB ObjectId using the isValidObjectId function from the Mongoose library. This function is often used in routes where the request parameter is expected to be an

ObjectId, ensuring that only valid ObjectIds are accepted, and it throws an error if an invalid ObjectId is provided.

Method:

1. **checkObjectId(req,res,next):**
    - the checkObjectId middleware ensures that the value of req.params.id is a valid MongoDB ObjectId. If it's not valid, it returns a 404 status and throws an error; otherwise, it allows the request to continue to the next middleware or route. This type of validation is useful in scenarios where route parameters are expected to represent MongoDB ObjectId values.

Class 4: errorMiddleware



Description: these middleware functions help manage the application's error-handling process. The notFound middleware is invoked when a route is not defined, setting a 404 status and passing an error to the next middleware. The errorHandler middleware responds to errors by determining an appropriate status code, extracting error information, and sending a JSON response with the error details. These middleware functions are often included in the main Express application file to handle unexpected scenarios.

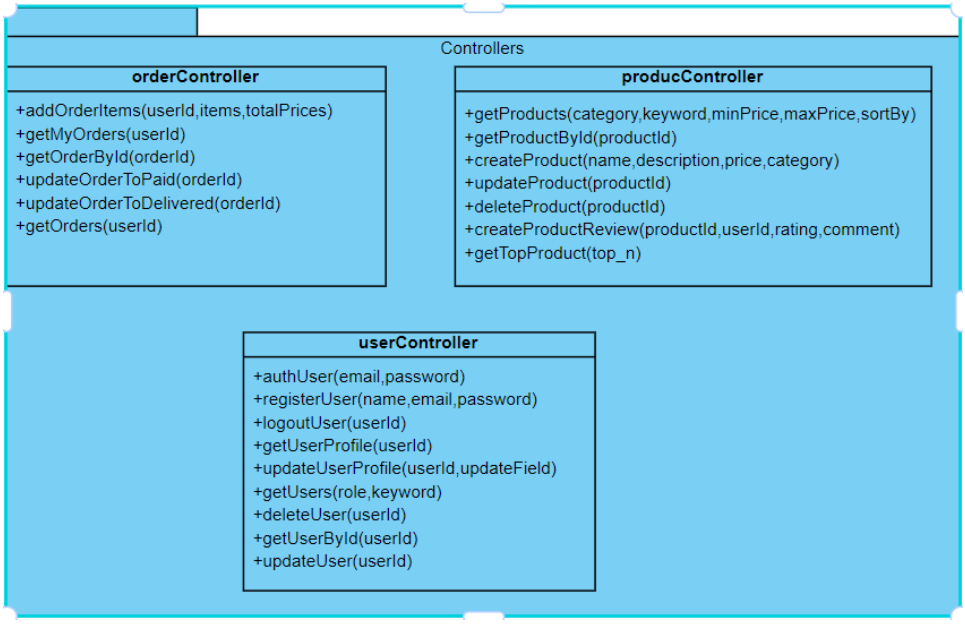
Methods:

1. **notFound(req,res,next):**
    - This middleware is designed to handle requests to routes that are not defined in the application. It creates a new Error object with a message indicating that the requested resource was not found (Not Found - ${req.originalUrl}).
    - It then sets the HTTP status code to 404 and passes the error to the next middleware in the stack.
2. **errorHandler(err,req,res,next):**
    - This middleware is responsible for handling errors that occur during the processing of a request. It takes four parameters: err (the error object), req (Express request object), res (Express

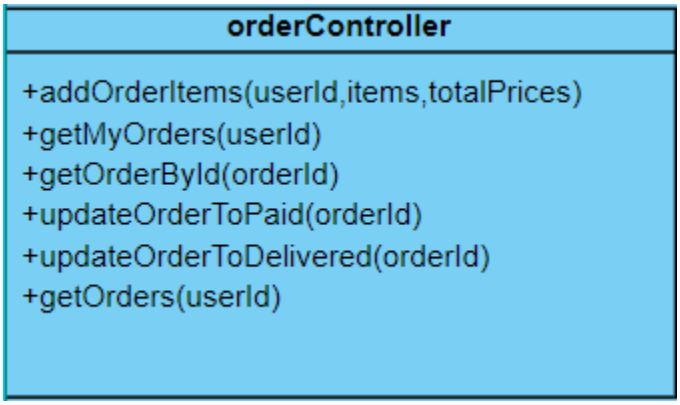
response object), and next (Express next middleware function).

- It determines the appropriate HTTP status code based on the existing response status code (res.statusCode). If the status code is 200 (OK), it sets the status code to 500 (Internal Server Error); otherwise, it uses the existing status code.
- The error message is extracted from the error object, and a JSON response is sent with the error message and, in non-production environments, the full error stack.

Key classes of Package Controller



Class 1: orderController



Description: this class sets up an API for managing orders in an e-commerce system, with features like order creation, payment verification, and order status updates. It follows a modular structure with separate controllers for each functionality, making the codebase organized and maintainable.

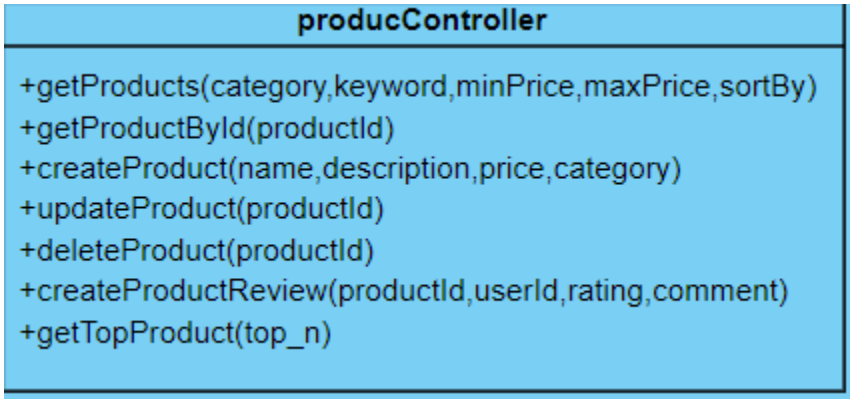
Method:

1. **addOrderItems(userId,items,totalPrices):**
   - This function creates a new order in the e-commerce system. It takes three parameters - userId representing the ID of the user placing the order, items containing details of the items being ordered, and totalPrices representing the total price of the order. The function fetches product details from the database, ensures the correctness of prices, calculates the order prices, and creates a new order in the database. It returns the newly created order.
2. **GetMyOrders(userId):**
   - This function retrieves orders for a logged-in user. It takes the userId parameter to identify the user whose orders are to be fetched. The function queries the database to find all orders associated with the specified user and returns the list of orders.
3. **GetOrderById(orderId):**
   - This function retrieves a specific order by its unique ID. It takes the orderId parameter, queries the database for the corresponding order, and returns the order details. If no order is found, it throws an error with a status of 404 and a message indicating that the order was not found.
4. **UpdateOrderToPaid(orderId):**
   - This function updates the status of an order to "paid" after verifying a PayPal payment. It takes the orderId parameter, verifies the PayPal payment associated with the order, checks for duplicate transactions, and updates the order status accordingly. If the payment is not verified or if the transaction has been used before, it throws an error. It returns the updated order details if successful.
5. **UpdateOrderToDelivered(orderId):**
   - This function marks an order as "delivered." It takes the orderId parameter, queries the database to find the corresponding order, and updates the order status to indicate that it has been delivered. This function is

accessible only to administrators. It returns the updated order details if successful.

6. **GetOrders(userId):**
   - This function retrieves all orders, and it is intended for use by administrators. It takes the userId parameter, queries the database to find all orders, and returns a list of orders with additional user information (ID and name) populated. This function is accessible only to users with administrator privileges.

Cass 2: productController



Description: These functions collectively provide a comprehensive set of functionalities for managing products, including retrieval, creation, updating, deletion, reviews, trending status, comments

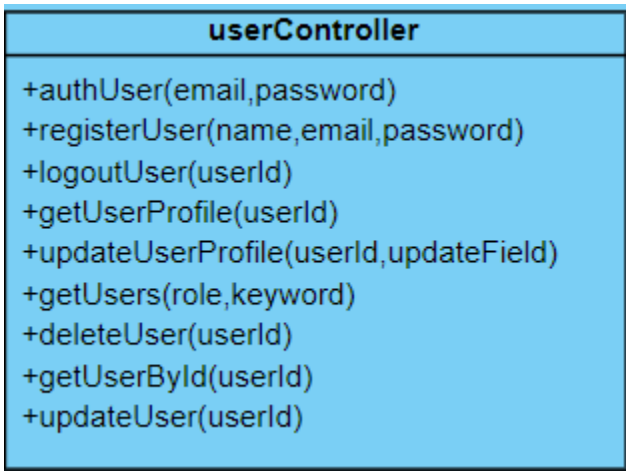
Method:

1. **GetProducts(category,keyword,minPrice,maxPrice,sortBy):**
   - Retrieves a list of products based on optional parameters such as category, keyword search, price range (minimum and maximum), and sorting criteria. The function queries the database with the specified filters and returns a paginated list of products meeting the criteria.
2. **GetProductById(productId):**
   - Retrieves a single product by its unique identifier (ID). The function queries the database for the specified product using the provided ID and returns its details. If the product is not found, it throws an error with a status of 404 and a message indicating that the product was not found.
3. **CreateProduct(name,description,price,category):**
   - Creates a new product with the specified name, description, price, and category. The function sets default values for other properties, such as user, image, brand, countInStock, numReviews, and description. The newly created product is saved to the database, and its details are returned in the response.

4. **UpdateProduct(productId):**
    - Updates an existing product with new information. The function takes updated product details from the request body, queries the database for the specified product by ID, and modifies its properties. The updated product is then saved to the database, and its details are returned in the response
5. **DeleteProduct(productId):**
    - Deletes an existing product identified by the provided product ID. The function queries the database for the specified product and, if found, removes it. The response includes a message indicating that the product has been removed. If the product is not found, it throws an error with a status of 404.
6. **CreateProductReview(productId,userId,rating,comment):**
    - Allows a user to add a review for a specific product. The function checks if the user has already reviewed the product and, if not, adds the new review to the product's reviews array. The overall rating and review count for the product are updated accordingly.
7. **GetTopProduct(top_n)**
    - Retrieves a list of the top-rated products. The function queries the database for all products, sorts them in descending order based on their ratings, and returns the top N products as specified by the top_n parameter.

Class 3: userController

Description: This class provides a comprehensive set of user authentication and management endpoints for a web application with appropriate error handling and access controls.



Method:

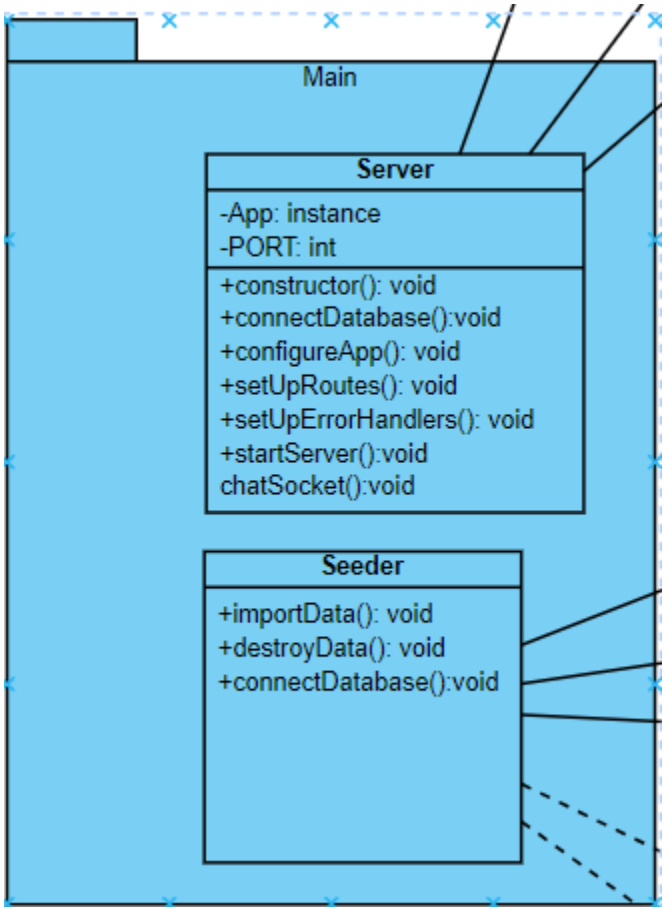
1. **AuthUser(email,password):**
    - This function handles user authentication. It expects an email and password as parameters.
    - It queries the database to find a user with the provided email.

- If a user is found and the password matches, it generates a token using the user's ID and sends back user information in the response, excluding sensitive data like the password.
- If authentication fails, it returns a 401 Unauthorized status with an error message.

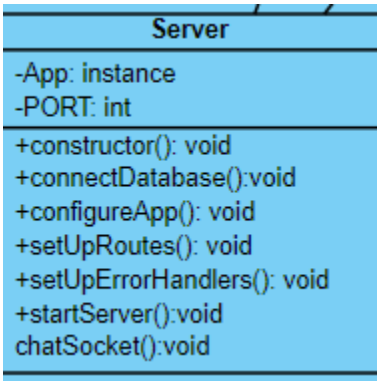
2. **RegisterUser(name,email,password)**
   - This function handles user registration. It expects a user's name, email, and password as parameters.
   - It checks if a user with the provided email already exists in the database. If so, it returns a 400 Bad Request status with an error message.
   - If the email is unique, it creates a new user using the User model, generates an authentication token, and sends back user information in the response.
   - If user creation fails, it returns a 400 Bad Request status with an error message.
3. **LogoutUser(userId)**
   - This function handles user logout by clearing the JWT (JSON Web Token) cookie.
   - It sends a response with a message of success.
4. **GetUserProfile(userId)**
   - This function retrieves the user's profile based on the user ID stored in the request.
   - If the user is found, it sends back the user's information in the response.
   - If the user is not found, it returns a 404 Not Found status with an error message.
5. **UpdateUserProfile(userId,updateField)**
   - This function updates the user's profile (name, email, and optionally password) based on the authenticated user.
   - It retrieves the user by ID, updates the specified fields, and saves the updated user.
   - It then sends back the updated user information in the response.
   - If the user is not found, it returns a 404 Not Found status with an error message.
6. **GetUsers(role,keyword)**
   - This function retrieves all users from the database (admin access required).
   - It sends back an array of user objects in the response.
7. **DeleteUser(userId)**
   - This function deletes a user by their ID (admin access required), except for admin users.
   - If the user is an admin, it returns a 400 Bad Request status with an error message.
   - If the user is found and not an admin, it deletes the user and sends back a success message.
   - If the user is not found, it returns a 404 Not Found status with an error message.
8. **GetUserById(userId)**
   - This function retrieves a user by ID (admin access required), excluding the password.
   - If the user is found, it sends back the user information in the response.
   - If the user is not found, it returns a 404 Not Found status with an error message.
9. **UpdateUser(userId)**
   - This function updates a user's information (name, email, and admin status) based on the provided user ID (admin access required).
   - It retrieves the user by ID, updates the specified fields, and saves the updated user.
   - It then sends back the updated user information in the response.

If the user is not found, it returns a 404 Not Found status with an error message.

Key classes of Package: Main



Class 1: Server

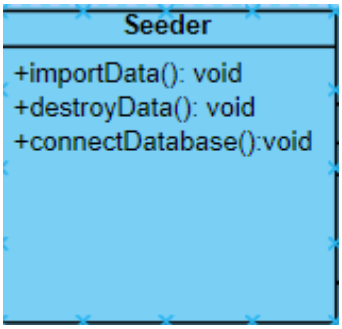


- **Description:** to provide a backend server for the application.
- **Method:**
  - **constructor() : void**

- to creates an instance of the Express.js application.

- **connectDatabase() : void**
  - to establish a connection to the MongoDB database.
- **configureApp() : void**
  - to configure the Express app, including middleware for JSON and URL-encoded request bodies, as well as cookie parsing.
- **setUpRoutes() : void**
  - to imported route handlers with their respective prefixes.
- **setUpErrorHandlers() : void**
  - to handle 404 (Not Found) and other errors.
- **startServer() : void**
  - to starts the server on the specified port and logs a message indicating the mode and the port.
- **chatSocket() : void**
  - to set up a Socket.IO server that handles user connections, room joining, chat messages, and disconnections.

Class 2: Seeder



Description: import and destroy data in a MongoDB database using Mongoose.

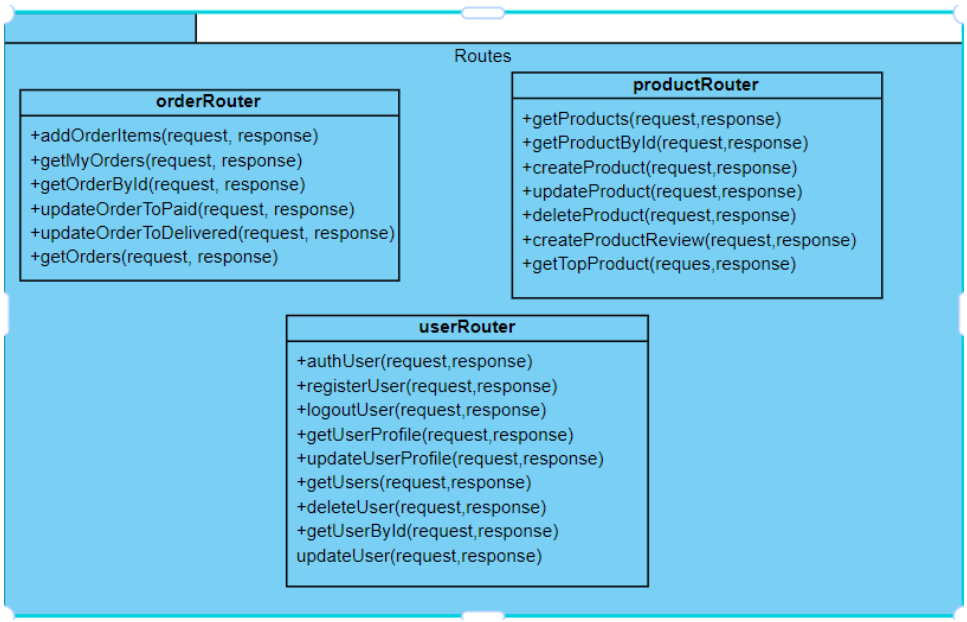
Method:

- importData() : void
  - to delete all existing documents in the 'Order', 'Product', and 'User' collections.
  - Inserts sample users into the User collection and retrieves the ID of the first user (presumably an admin user).
  - Modifies each product in the products array by adding the admin user's ID as the user field.Inserts the modified products into the Product collection.
- destroyData() : void

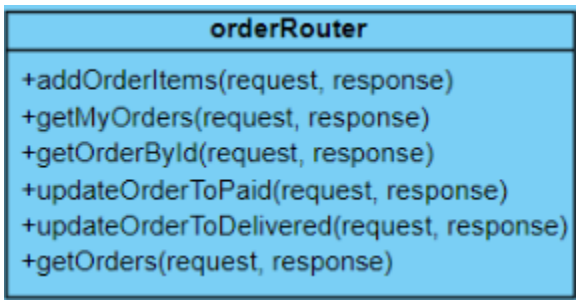
- to delete all existing documents in the 'Order', 'Product', and 'User' collections.

- connectDatabase() : void
  - to establish a connection to the MongoDB database.

Key classes of Package: Routes



Class 1: orderRouter



Description:

This class defines an Express router for handling routes related to orders in a web application. It uses the Express framework and includes routes for creating, retrieving, and updating orders. The routes are associated with corresponding controller functions from the orderController.js file, and they include middleware functions (protect and admin) from the authMiddleware.js file to handle user authentication and authorization.

Method:

1. addOrderItems(request,response):

- This method handles the creation of new orders by adding order items to the database. It is typically associated with a route that receives a POST request containing the details of the items to be ordered. The method validates the incoming data, associates the order with the authenticated user (using protect middleware), and then saves the order details, including the items and their quantities, in the database.
- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the order creation.

2. GetMyOrders(request,response):

- This method retrieves the orders associated with the authenticated user. It is typically linked to a route that handles a GET request for fetching the orders of the currently logged-in user. The method queries the database for orders with the user's ID and returns the order details.
- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the retrieval of user-specific orders.

3. GetOrderById(request,response):

- This method retrieves a specific order by its unique identifier (ID). It is typically associated with a route that handles a GET request, where the order ID is provided as a parameter in the request URL. The method validates the order ID, queries the database for the corresponding order, and returns its details.
- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the retrieval of the specific order.

4. UpdateOrderToPaid(request,response):

- This method updates the payment status of a specific order. It is usually linked to a route that handles a PUT request to mark an order as paid. The method validates the order ID, retrieves the order from the database, updates its payment status, and saves the changes.
- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the update of the payment status.

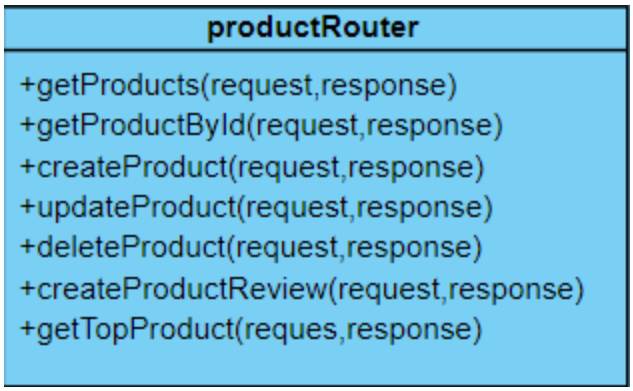
5. UpdateOrderToDelivered(request,response):

- This method updates the delivery status of a specific order. It is typically associated with a route that handles a PUT request to mark an order as delivered. The method validates the order ID, checks for admin authorization (using admin middleware), retrieves the order from the database, updates its delivery status, and saves the changes.
- Middleware Used: protect and admin middlewares are used to ensure that the request is authenticated and authorized as an admin before allowing the update of the delivery status.

6. GetOrders(request,response):

- This method retrieves all orders from the database. It is usually linked to a route that handles a GET request to fetch all orders. The method checks for admin authorization (using admin middleware), queries the database for all orders, and returns their details.
- Middleware Used: protect and admin middlewares are used to ensure that the request is authenticated and authorized as an admin before allowing the retrieval of all orders.

Class 2: productRoutes



Description: This class sets up routes for handling various actions related to products in an Express application. These routes include creating, retrieving, updating, and deleting products, as well as managing product reviews, trending products, and comments. The actual implementation of these actions is expected to be in the productController.js file.

Method:

1. GetProducts(request,response):

- Retrieves a list of products. This route is responsible for fetching and returning a list of products from your database. It can be used to display products on the client side.

2. GetProductById(request,response):

- Retrieves a specific product by its unique identifier (ID). This route fetches and returns the details of a specific product identified by the provided product ID. The checkObjectId middleware ensures that the ID is in the correct format before querying the database.
- Uses the checkObjectId middleware to validate the format of the product ID.

3. CreateProduct(request,response):

- Creates a new product. This route is responsible for creating a new product based on the data provided in the request body. It requires authentication (protect middleware) and checks if the user has seller privileges (seller middleware) before allowing the creation of a new product.
- Uses the protect middleware to ensure the user is authenticated, and the seller middleware to ensure the user has seller privileges.

4. UpdateProduct(request,response):

- Updates information for a specific product. This route updates the information of a specific product identified by the provided product ID. It requires authentication, seller privileges, and ensures that the product ID is in the correct format.
- Uses the protect middleware to ensure the user is authenticated, the seller middleware to ensure the user has seller privileges, and the checkObjectId middleware to validate the format of the product ID.

5. DeleteProduct(request,response):

- Deletes a specific product. This route deletes a product identified by the provided product ID. It requires authentication, seller privileges, and ensures that the product ID is in the correct format.
- Uses the protect middleware to ensure the user is authenticated, the seller middleware to ensure the user has seller privileges, and the checkObjectId middleware to validate the format of the product ID.

6. CreateProductReview(request,response):

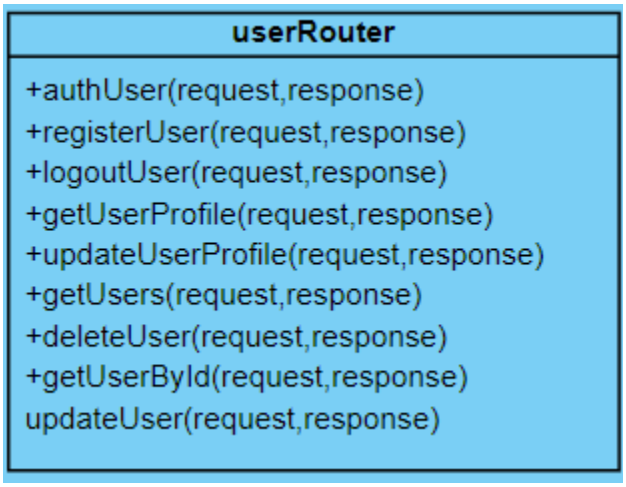
- Adds a review for a specific product. This route allows authenticated users to add reviews for a specific product. The checkObjectId middleware ensures that the product ID is in the correct format.

- Uses the protect middleware to ensure the user is authenticated and the checkObjectId middleware to validate the format of the product ID.

7. GetTopProduct(request,response)

- Retrieves a list of top-rated products. This route fetches and returns a list of top-rated products. The definition of "top-rated" may depend on the criteria you have set for ranking products.

Class 3: userRouter



Description:

This code sets up routes for user registration, authentication, logout, profile management, listing users, and performing administrative actions (like deleting, updating, and retrieving user details) with appropriate authentication and authorization checks. The middleware functions (protect and admin) play a crucial role in securing these routes.

Method:

1. AuthUser(request,response):

- This function is responsible for authenticating users by checking their credentials, typically username/email and password. Upon successful authentication, it generates and returns an authentication token, allowing the user to make authenticated requests.

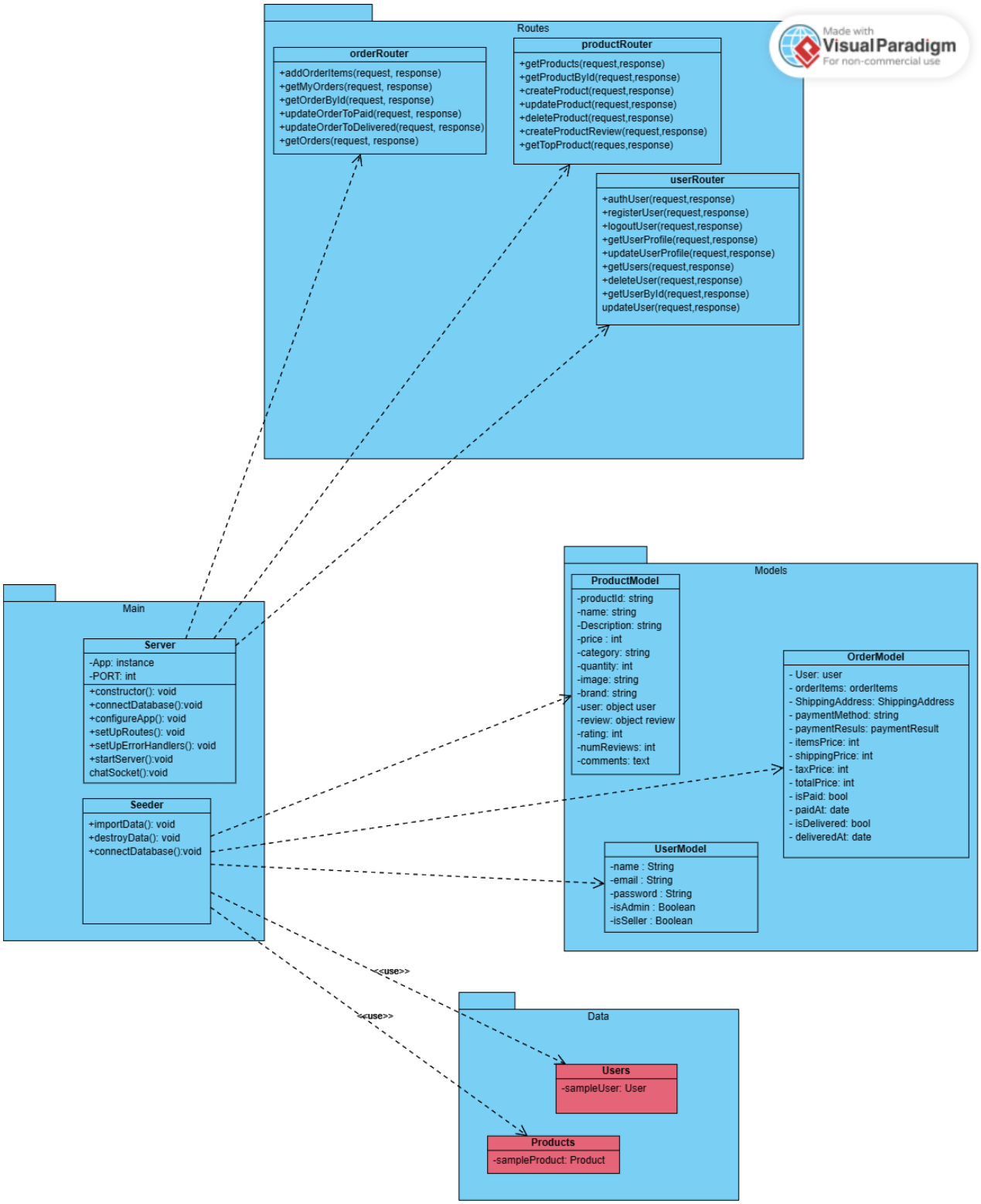
2. RegisterUser(request,response)

- Manages the registration process for new users. It typically involves validating user-provided data, creating a new user record in the database, and returning relevant information about the newly registered user.

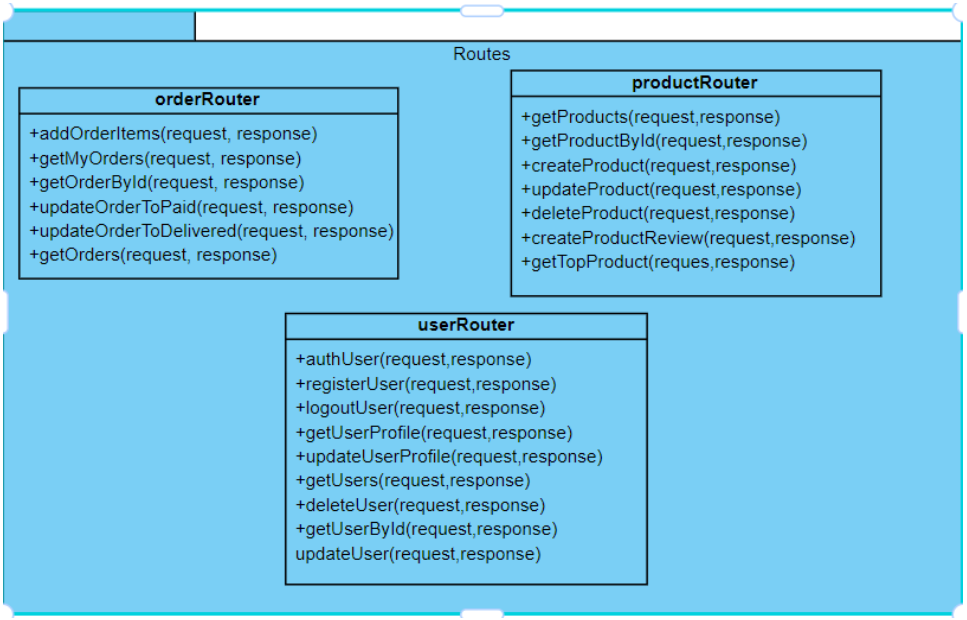
3. **LogoutUser(request,response)**
    - Handles the user logout process, which often includes invalidating the user's authentication token. This action helps enhance the security of the user's account by preventing unauthorized access.
4. **GetUserProfile(request,response)**
    - Retrieves the profile information of the currently authenticated user. This can include details such as username, email, and any additional user-specific information.
    - Uses the protect middleware to ensure that the user is authenticated before allowing access to their profile.
5. **UpdateUserProfile(request,response)**
    - Allows the currently authenticated user to update their profile information. This can include changing the username, email, or other user-specific details.
    - Middleware: Uses the protect middleware to ensure that the user is authenticated before allowing them to update their profile.
6. **GetUsers(request,response)**
    - Retrieves a list of users from the database. This functionality is typically restricted to users with administrative privileges to maintain user privacy.
    - Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.
7. **DeleteUser(request,response)**
    - Deletes a user by their ID. This operation is typically restricted to users with administrative privileges to manage user accounts.
    - Middleware: Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.
8. **GetUserById(request,response)**
    - Retrieves detailed information about a specific user based on their user ID. This functionality is typically used by administrators to view specific user details.
    - Middleware: Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.
9. **UpdateUser(request,response)**

- Updates the information of a specific user based on their user ID. This functionality is typically used by administrators to modify user details.
- Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.

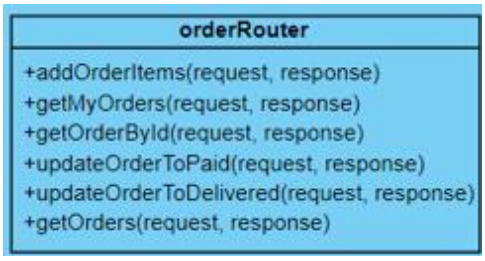
Class Diagram Of Main Back-end services: Main, Routes, Models, Data



Key class of package: Routes



Class 1: orderRouter



Description:

This class defines an Express router for handling routes related to orders in a web application. It uses the Express framework and includes routes for creating, retrieving, and updating orders. The routes are associated with corresponding controller functions from the orderController.js file, and they include middleware functions (protect and admin) from the authMiddleware.js file to handle user authentication and authorization.

Method:

7. addOrderItems(request,response):

- This method handles the creation of new orders by adding order items to the database. It is typically associated with a route that receives a POST request containing the details of the items to be ordered. The

method validates the incoming data, associates the order with the authenticated user (using protect middleware), and then saves the order details, including the items and their quantities, in the database.

- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the order creation.

8. GetMyOrders(request,response):

- This method retrieves the orders associated with the authenticated user. It is typically linked to a route that handles a GET request for fetching the orders of the currently logged-in user. The method queries the database for orders with the user's ID and returns the order details.
- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the retrieval of user-specific orders.

9. GetOrderById(request,response):

- This method retrieves a specific order by its unique identifier (ID). It is typically associated with a route that handles a GET request, where the order ID is provided as a parameter in the request URL. The method validates the order ID, queries the database for the corresponding order, and returns its details.
- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the retrieval of the specific order.

10. UpdateOrderToPaid(request,response):

- This method updates the payment status of a specific order. It is usually linked to a route that handles a PUT request to mark an order as paid. The method validates the order ID, retrieves the order from the database, updates its payment status, and saves the changes.
- Middleware Used: protect middleware is used to ensure that the request is authenticated before allowing the update of the payment status.

11. UpdateOrderToDelivered(request,response):

- This method updates the delivery status of a specific order. It is typically associated with a route that handles a PUT request to mark an order as delivered. The method validates the order ID, checks for admin authorization (using admin middleware), retrieves the

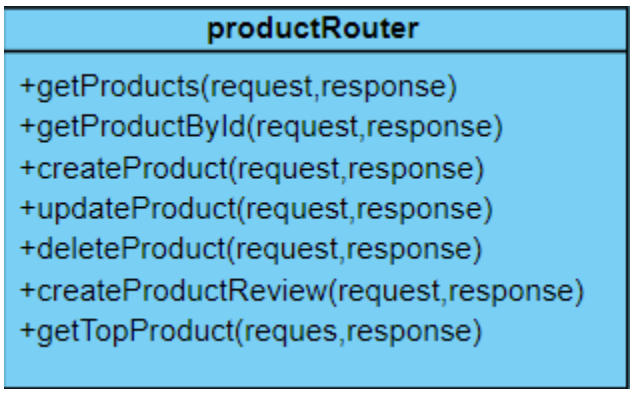
order from the database, updates its delivery status, and saves the changes.

- Middleware Used: protect and admin middlewares are used to ensure that the request is authenticated and authorized as an admin before allowing the update of the delivery status.

12. **GetOrders(request,response):**

- This method retrieves all orders from the database. It is usually linked to a route that handles a GET request to fetch all orders. The method checks for admin authorization (using admin middleware), queries the database for all orders, and returns their details.
- Middleware Used: protect and admin middlewares are used to ensure that the request is authenticated and authorized as an admin before allowing the retrieval of all orders.

Class 2: productRoutes



Description: This class sets up routes for handling various actions related to products in an Express application. These routes include creating, retrieving, updating, and deleting products, as well as managing product reviews, trending products, and comments. The actual implementation of these actions is expected to be in the productController.js file.

Method:

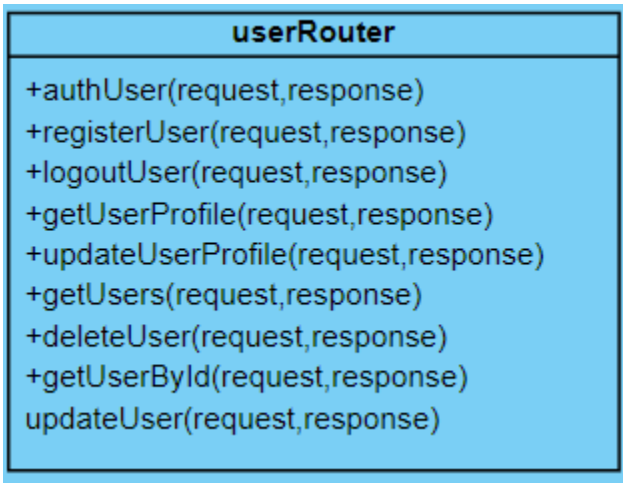
8. **GetProducts(request,response):**
   - Retrieves a list of products. This route is responsible for fetching and returning a list of products from your database. It can be used to display products on the client side.
9. **GetProductById(request,response):**
   - Retrieves a specific product by its unique identifier (ID). This route fetches and returns the details of a specific product identified by the provided product ID. The checkObjectId

middleware ensures that the ID is in the correct format before querying the database.

- Uses the checkObjectId middleware to validate the format of the product ID.

10. **CreateProduct(request,response):**
    - Creates a new product. This route is responsible for creating a new product based on the data provided in the request body. It requires authentication (protect middleware) and checks if the user has seller privileges (seller middleware) before allowing the creation of a new product.
    - Uses the protect middleware to ensure the user is authenticated, and the seller middleware to ensure the user has seller privileges.
11. **UpdateProduct(request,response):**
    - Updates information for a specific product. This route updates the information of a specific product identified by the provided product ID. It requires authentication, seller privileges, and ensures that the product ID is in the correct format.
    - Uses the protect middleware to ensure the user is authenticated, the seller middleware to ensure the user has seller privileges, and the checkObjectId middleware to validate the format of the product ID.
12. **DeleteProduct(request,response):**
    - Deletes a specific product. This route deletes a product identified by the provided product ID. It requires authentication, seller privileges, and ensures that the product ID is in the correct format.
    - Uses the protect middleware to ensure the user is authenticated, the seller middleware to ensure the user has seller privileges, and the checkObjectId middleware to validate the format of the product ID.
13. **CreateProductReview(request,response):**
    - Adds a review for a specific product. This route allows authenticated users to add reviews for a specific product. The checkObjectId middleware ensures that the product ID is in the correct format.
    - Uses the protect middleware to ensure the user is authenticated and the checkObjectId middleware to validate the format of the product ID.
14. **GetTopProduct(request,response)**
    - Retrieves a list of top-rated products. This route fetches and returns a list of top-rated products. The definition of "top-rated" may depend on the criteria you have set for ranking products.

Class 3: userRouter



Description:

This code sets up routes for user registration, authentication, logout, profile management, listing users, and performing administrative actions (like deleting, updating, and retrieving user details) with appropriate authentication and authorization checks. The middleware functions (protect and admin) play a crucial role in securing these routes.

Method:

10. **AuthUser(request,response):**
    - This function is responsible for authenticating users by checking their credentials, typically username/email and password. Upon successful authentication, it generates and returns an authentication token, allowing the user to make authenticated requests.
11. **RegisterUser(request,response)**
    - Manages the registration process for new users. It typically involves validating user-provided data, creating a new user record in the database, and returning relevant information about the newly registered user.
12. **LogoutUser(request,response)**
    - Handles the user logout process, which often includes invalidating the user's authentication token. This action helps enhance the security of the user's account by preventing unauthorized access.
13. **GetUserProfile(request,response)**
    - Retrieves the profile information of the currently authenticated user. This can include details such as

username, email, and any additional user-specific information.

- Uses the protect middleware to ensure that the user is authenticated before allowing access to their profile.

14. **UpdateUserProfile(request,response)**
    - Allows the currently authenticated user to update their profile information. This can include changing the username, email, or other user-specific details.
    - Middleware: Uses the protect middleware to ensure that the user is authenticated before allowing them to update their profile.
15. **GetUsers(request,response)**
    - Retrieves a list of users from the database. This functionality is typically restricted to users with administrative privileges to maintain user privacy.
    - Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.
16. **DeleteUser(request,response)**
    - Deletes a user by their ID. This operation is typically restricted to users with administrative privileges to manage user accounts.
    - Middleware: Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.
17. **GetUserById(request,response)**
    - Retrieves detailed information about a specific user based on their user ID. This functionality is typically used by administrators to view specific user details.
    - Middleware: Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.
18. **UpdateUser(request,response)**
    - Updates the information of a specific user based on their user ID. This functionality is typically used by administrators to modify user details.
    - Uses both the protect and admin middleware to ensure that the user making the request is authenticated and has administrative privileges.

Key classes of package: Main

Class 1: Server

- **Description:** to provide a backend server for the application.
- **Method:**
  - **constructor() : void**
    - to create an instance of the Express.js application.
  - **connectDatabase() : void**
    - to establish a connection to the MongoDB database.
  - **configureApp() : void**
    - to configure the Express app, including middleware for JSON and URL-encoded request bodies, as well as cookie parsing.
  - **setUpRoutes() : void**
    - to imported route handlers with their respective prefixes.
  - **setUpErrorHandlers() : void**
    - to handle 404 (Not Found) and other errors.
  - **startServer() : void**
    - to starts the server on the specified port and logs a message indicating the mode and the port.
  - **chatSocket() : void**
    - to set up a Socket.IO server that handles user connections, room joining, chat messages, and disconnections.
- **Communication**

Class 2: Seeder

Description: import and destroy data in a MongoDB database using Mongoose.

Method:

- importData() : void
  - to delete all existing documents in the 'Order', 'Product', and 'User' collections.
  - Inserts sample users into the User collection and retrieves the ID of the first user (presumably an admin user).
  - Modifies each product in the products array by adding the admin user's ID as the user field.Inserts the modified products into the Product collection.
- destroyData() : void
  - to delete all existing documents in the 'Order', 'Product', and 'User' collections.

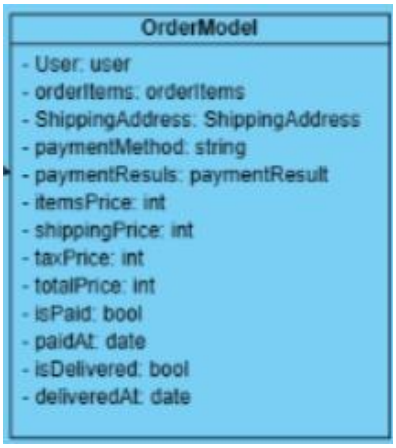
- connectDatabase() : void
  - to establish a connection to the MongoDB database.

Key classes of package: Models

Class 1: ProductModel

-
- **Description:** to provide Mongoose schemas for products and their reviews, and it creates a Mongoose model for products.
- **Method:** None

Class 2: OrderModel

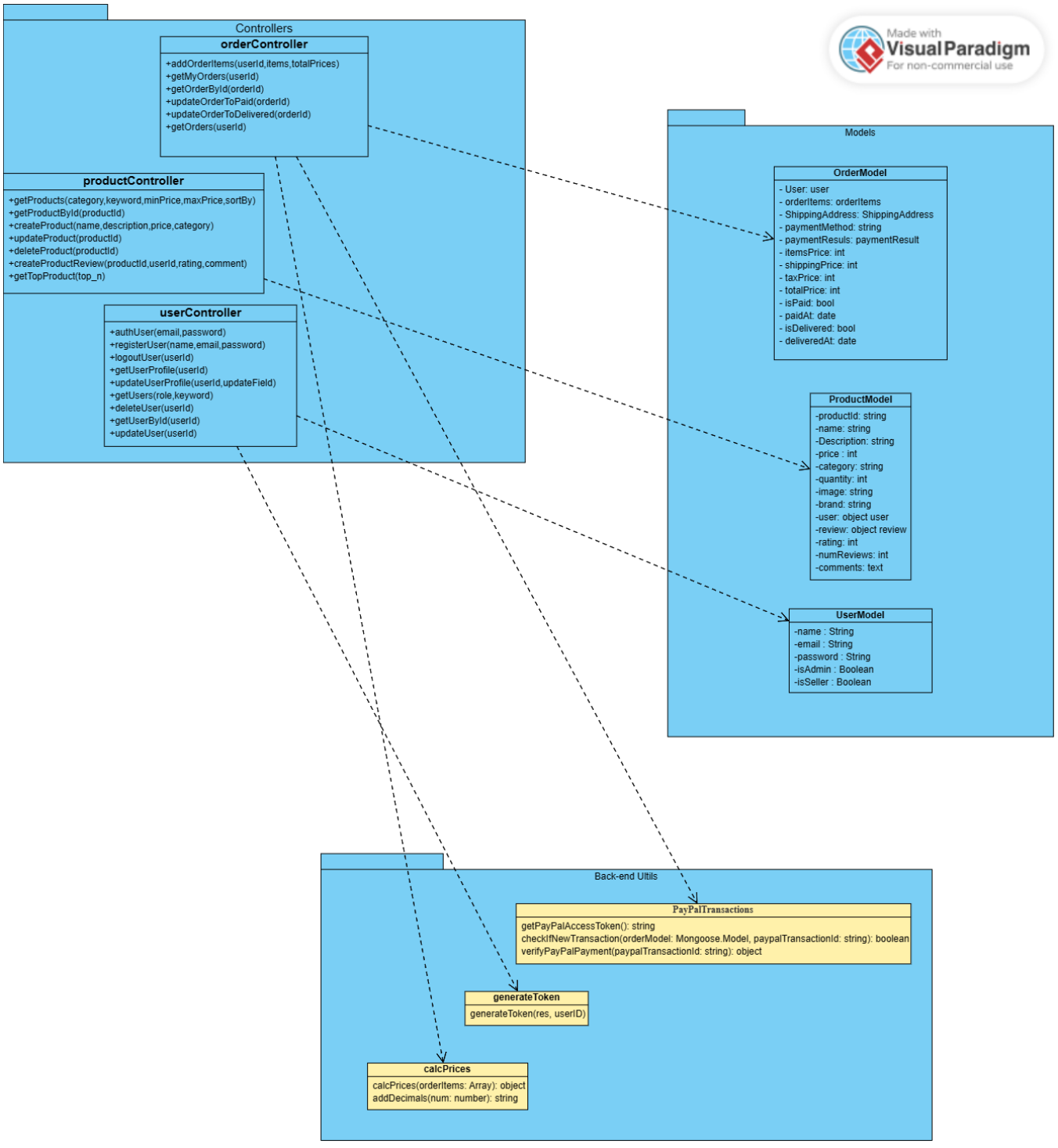


- **Description:** to establish a Mongoose schema for orders, providing a structure for storing order-related information in a MongoDB database.
- **Method:** None

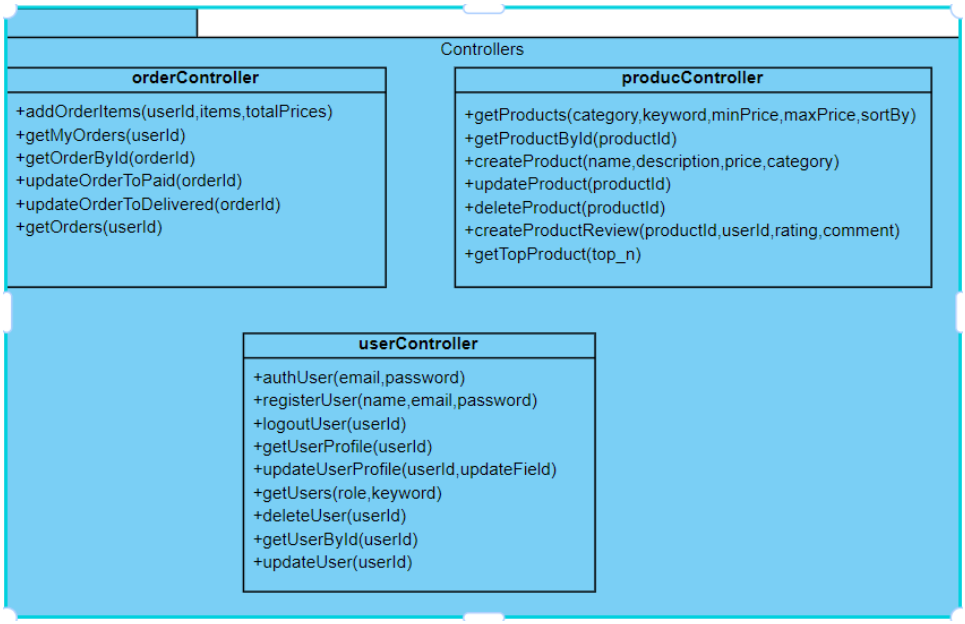
Class 3: UserModel

- **Description:** establishes a Mongoose schema for users, including fields for basic user information, incorporates methods for password validation and encryption.
- **Method:** None

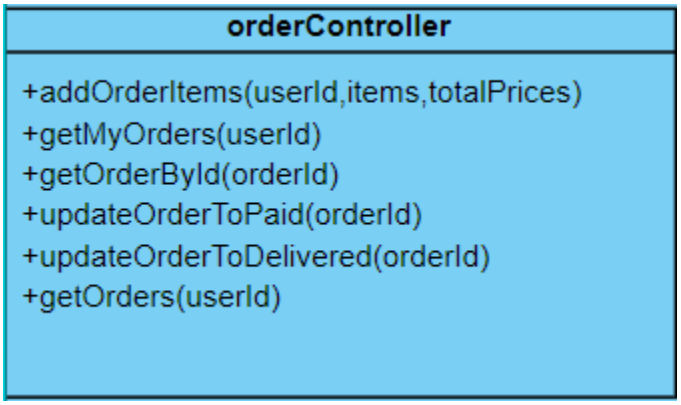
Class Diagram Of Controller Services: Controller, Back-end Utils, Model



Key classes of Package Controller



Class 1: orderController



Description: this class sets up an API for managing orders in an e-commerce system, with features like order creation, payment verification, and order status updates. It follows a modular structure with separate controllers for each functionality, making the codebase organized and maintainable.

Method:

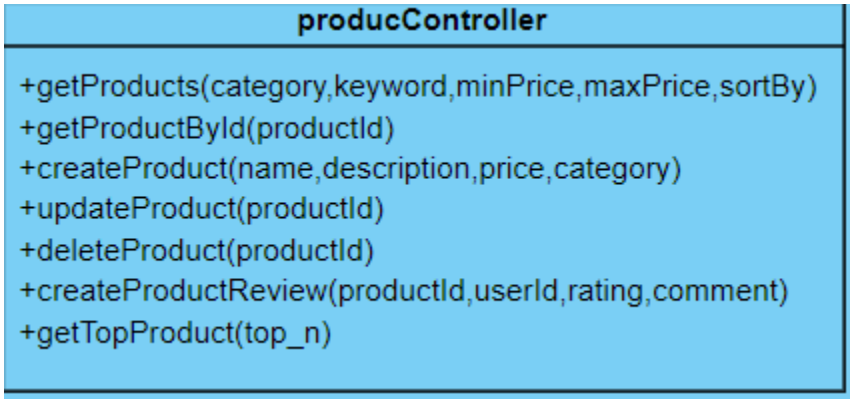
1. addOrderItems(userId,items,totalPrices):

- This function creates a new order in the e-commerce system. It takes three parameters - userId representing the ID of the user placing the order, items containing details of the items being ordered, and totalPrices representing the total price of the order. The function

fetches product details from the database, ensures the correctness of prices, calculates the order prices, and creates a new order in the database. It returns the newly created order.

2. **GetMyOrders(userId):**
   - This function retrieves orders for a logged-in user. It takes the userId parameter to identify the user whose orders are to be fetched. The function queries the database to find all orders associated with the specified user and returns the list of orders.
3. **GetOrderById(orderId):**
   - This function retrieves a specific order by its unique ID. It takes the orderId parameter, queries the database for the corresponding order, and returns the order details. If no order is found, it throws an error with a status of 404 and a message indicating that the order was not found.
4. **UpdateOrderToPaid(orderId):**
   - This function updates the status of an order to "paid" after verifying a PayPal payment. It takes the orderId parameter, verifies the PayPal payment associated with the order, checks for duplicate transactions, and updates the order status accordingly. If the payment is not verified or if the transaction has been used before, it throws an error. It returns the updated order details if successful.
5. **UpdateOrderToDelivered(orderId):**
   - This function marks an order as "delivered." It takes the orderId parameter, queries the database to find the corresponding order, and updates the order status to indicate that it has been delivered. This function is accessible only to administrators. It returns the updated order details if successful.
6. **GetOrders(userId):**
   - This function retrieves all orders, and it is intended for use by administrators. It takes the userId parameter, queries the database to find all orders, and returns a list of orders with additional user information (ID and name) populated. This function is accessible only to users with administrator privileges.

Class 2: productController



Description: These functions collectively provide a comprehensive set of functionalities for managing products, including retrieval, creation, updating, deletion, reviews, trending status, comments

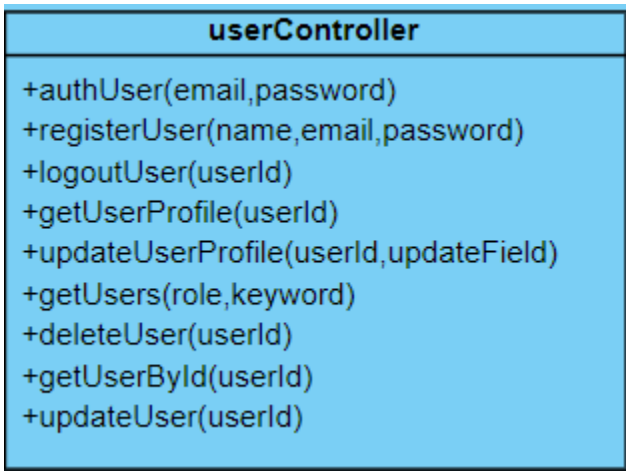
Method:

1. **GetProducts(category,keyword,minPrice,maxPrice,sortBy):**
   - Retrieves a list of products based on optional parameters such as category, keyword search, price range (minimum and maximum), and sorting criteria. The function queries the database with the specified filters and returns a paginated list of products meeting the criteria.
2. **GetProductById(productId):**
   - Retrieves a single product by its unique identifier (ID). The function queries the database for the specified product using the provided ID and returns its details. If the product is not found, it throws an error with a status of 404 and a message indicating that the product was not found.
3. **CreateProduct(name,description,price,category):**
   - Creates a new product with the specified name, description, price, and category. The function sets default values for other properties, such as user, image, brand, countInStock, numReviews, and description. The newly created product is saved to the database, and its details are returned in the response.
4. **UpdateProduct(productId):**
   - Updates an existing product with new information. The function takes updated product details from the request body, queries the database for the specified product by ID, and modifies its properties. The updated product is then saved to the database, and its details are returned in the response
5. **DeleteProduct(productId):**
   - Deletes an existing product identified by the provided product ID. The function queries the database for the specified product and, if found, removes it. The response includes a message indicating that the product has been removed. If the product is not found, it throws an error with a status of 404.
6. **CreateProductReview(productId,userId,rating,comment):**
   - Allows a user to add a review for a specific product. The function checks if the user has already reviewed the product and, if not, adds the new review to the product's reviews array. The overall rating and review count for the product are updated accordingly.

7. GetTopProduct(top_n)

- Retrieves a list of the top-rated products. The function queries the database for all products, sorts them in descending order based on their ratings, and returns the top N products as specified by the top_n parameter.

Class 3: userController



Description: This class provides a comprehensive set of user authentication and management endpoints for a web application with appropriate error handling and access controls.

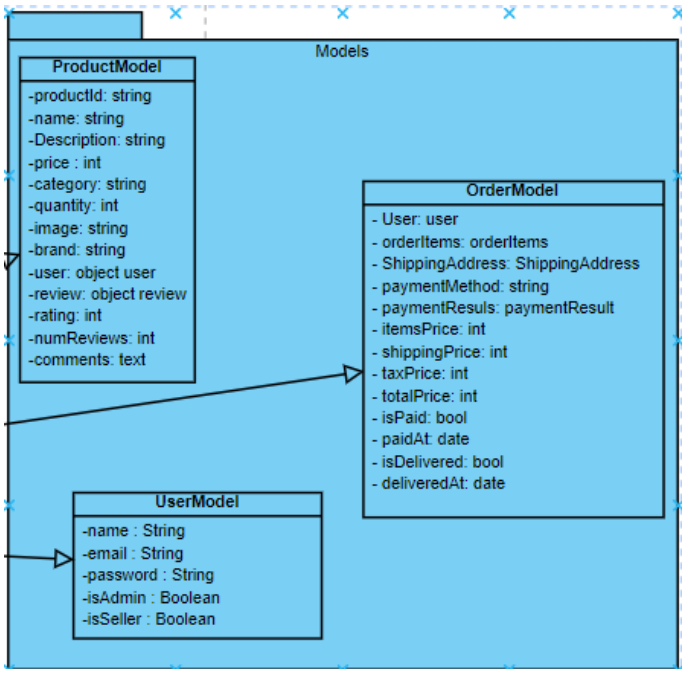
Method:

1. **AuthUser(email,password):**
   - This function handles user authentication. It expects an email and password as parameters.
   - It queries the database to find a user with the provided email.
   - If a user is found and the password matches, it generates a token using the user's ID and sends back user information in the response, excluding sensitive data like the password.
   - If authentication fails, it returns a 401 Unauthorized status with an error message.
2. **RegisterUser(name,email,password)**
   - This function handles user registration. It expects a user's name, email, and password as parameters.
   - It checks if a user with the provided email already exists in the database. If so, it returns a 400 Bad Request status with an error message.
   - If the email is unique, it creates a new user using the User model, generates an authentication token, and sends back user information in the response.
   - If user creation fails, it returns a 400 Bad Request status with an error message.
3. **LogoutUser(userId)**
   - This function handles user logout by clearing the JWT (JSON Web Token) cookie.
   - It sends a response with a message of success.
4. **GetUserProfile(userId)**
   - This function retrieves the user's profile based on the user ID stored in the request.

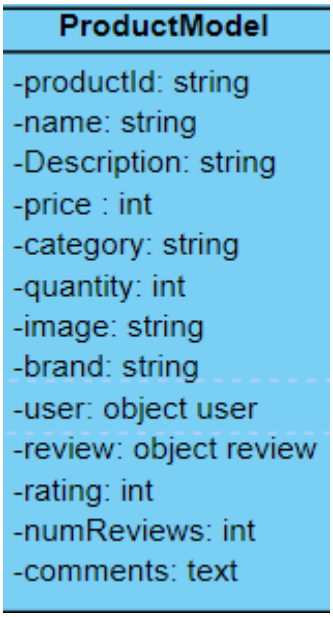
- If the user is found, it sends back the user's information in the response.
- If the user is not found, it returns a 404 Not Found status with an error message.

5. **UpdateUserProfile(userId,updateField)**
   - This function updates the user's profile (name, email, and optionally password) based on the authenticated user.
   - It retrieves the user by ID, updates the specified fields, and saves the updated user.
   - It then sends back the updated user information in the response.
   - If the user is not found, it returns a 404 Not Found status with an error message.
6. **GetUsers(role,keyword)**
   - This function retrieves all users from the database (admin access required).
   - It sends back an array of user objects in the response.
7. **DeleteUser(userId)**
   - This function deletes a user by their ID (admin access required), except for admin users.
   - If the user is an admin, it returns a 400 Bad Request status with an error message.
   - If the user is found and not an admin, it deletes the user and sends back a success message.
   - If the user is not found, it returns a 404 Not Found status with an error message.
8. **GetUserById(userId)**
   - This function retrieves a user by ID (admin access required), excluding the password.
   - If the user is found, it sends back the user information in the response.
   - If the user is not found, it returns a 404 Not Found status with an error message.
9. **UpdateUser(userId)**
   - This function updates a user's information (name, email, and admin status) based on the provided user ID (admin access required).
   - It retrieves the user by ID, updates the specified fields, and saves the updated user.
   - It then sends back the updated user information in the response.
   - If the user is not found, it returns a 404 Not Found status with an error message.
10.

Key classes of package: Models

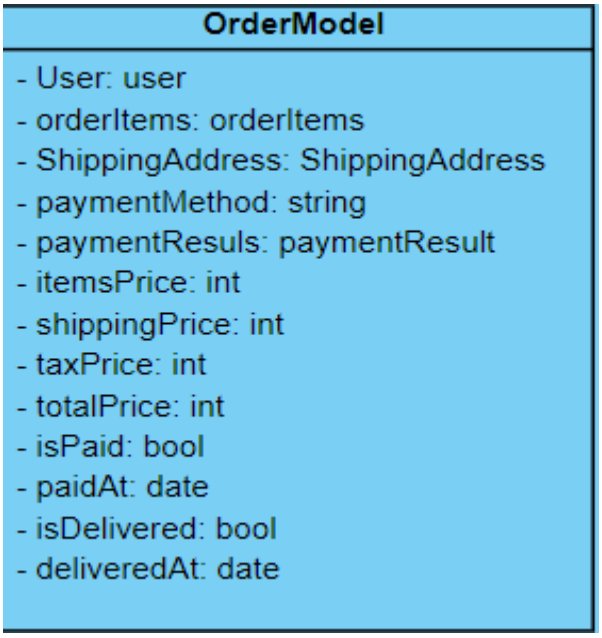


Class 1: ProductModel



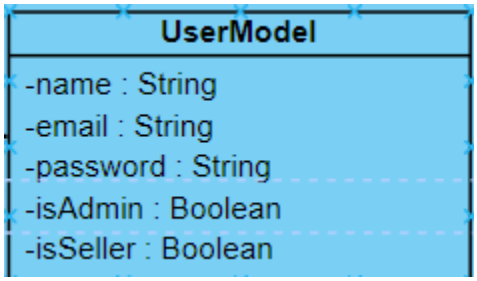
- **Description:** to provide Mongoose schemas for products and their reviews, and it creates a Mongoose model for products.
- **Method:** None

Class 2: OrderModel



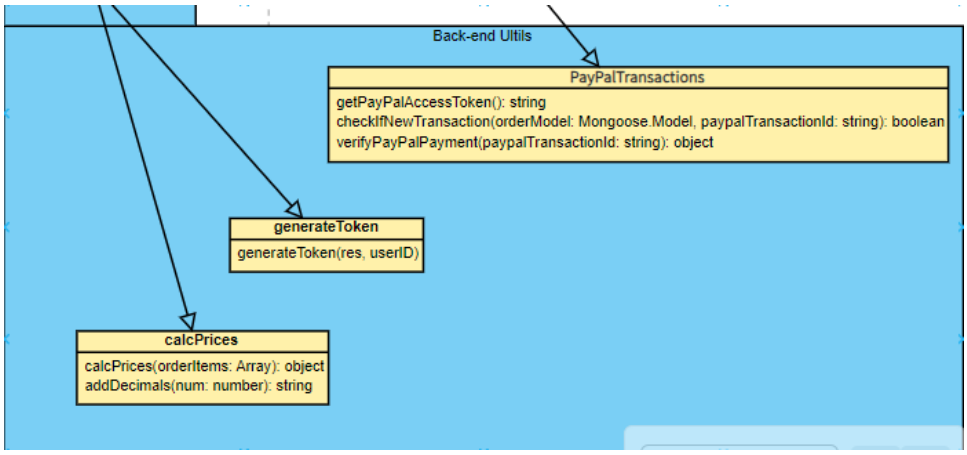
- **Description:** to establish a Mongoose schema for orders, providing a structure for storing order-related information in a MongoDB database.
- **Method:** None

Class 3: UserModel

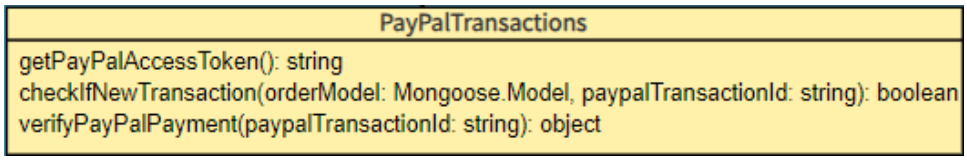


- **Description:** establishes a Mongoose schema for users, including fields for basic user information, incorporates methods for password validation and encryption.
- **Method:** None

Key classes of Package Backend utils



Class 1:PayPalTransactions



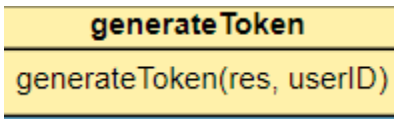
Description:

- The class use the fetch API for making HTTP requests to the PayPal API.
- The access token obtained in getPayPalAccessToken is used for authentication in subsequent requests.
- Error handling is included, and errors are logged to the console.
- The code assumes the existence of a MongoDB database and a Mongoose model for handling orders.

Method:

- GetPayPalAccessToken : Fetches an access token from the PayPal API using client credentials (client ID and app secret).
- checkIfNewTransaction: Checks if a PayPal transaction is new by querying a database (MongoDB) and comparing the transaction ID with existing orders.
- verifyPayPalPayment :Verifies a PayPal payment by making a request to the PayPal API with the transaction ID.

Class 2:generaToken

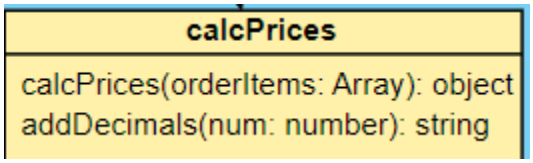


Description: It uses the jsonwebtoken library to create a JSON Web Token (JWT) and sets it as an HTTP-only cookie. The purpose of this code is typically for user authentication in web applications.

Method:

- **GenerateToken:** function is to facilitate user authentication by creating a secure token that can be used to identify and authorize users within a web application.

Class 3:calcPrices



Description: The code includes a note explaining that there was a change from the original course code to address a type coercion issue when calling addDecimals with a string argument. The addDecimals function is designed to receive a number and return a string, so it is adjusted to handle this expectation.

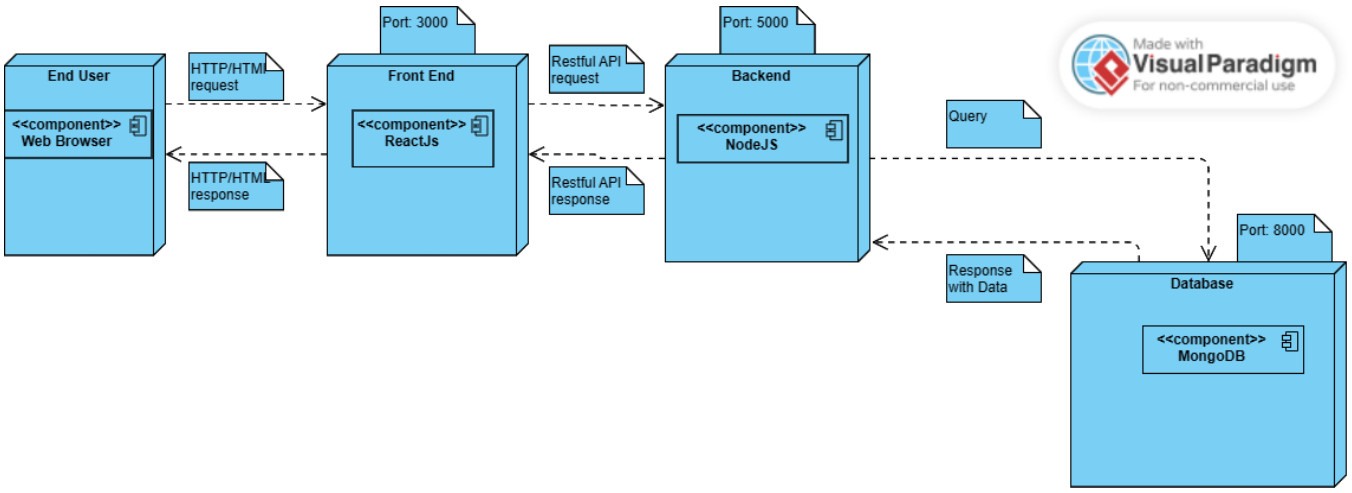
Method:

AddDecimals: The addDecimals function takes a numerical argument (num) and returns a string representation of that number with two decimal places.

CalcPrices: function calculates the prices for an array of order items. It considers the item price, shipping cost, tax amount, and total price. The prices are computed in whole numbers (pennies) to avoid potential issues with floating-point number calculations.

V. Deployment

Deployment-Diagram:



End User

- Description: The End User node represents the user of the e-commerce website. It's where the interaction with the website begins. The user navigates

the website using a web browser, which is represented as a component in this node.

- Relationship with:
    - Frontend: The End User node interacts directly with the Front-End node. It sends HTTP/HTML requests and receives HTTP/HTML responses. This is a bidirectional relationship, meaning data flows both ways.

Frontend

- Description: The Front-End node serves as the interface between the user and the server-side infrastructure. It is responsible for receiving user requests, processing them, and sending back the appropriate responses. This node hosts the ReactJS component, which is a popular JavaScript library for building user interfaces. The Front-End node is hosted on port 3000.
- Relationship with:
    - End-user: The Front-End node receives the HTTP/HTML request from the End User node and sends back an HTTP/HTML response.
    - Backend: The Front-End node sends a Restful API request to the Backend node and receives a Restful API response. This is also a bidirectional relationship.

Backend

- Description: The Backend is the server-side of the application. The Backend receives Restful API requests from the Front End, processes them, and queries the MongoDB database for the required data. Once the data is retrieved from the database, the Backend sends a response with the data back to the Front End. The Backend is also responsible for business logic, data validation, and authentication.
- Relationship with:
    - Frontend:
        - Backend waits for requests from Frontend, requests including actions like retrieving, creating, updating, or deleting data.
        - Backend communicates with Frontend through a defined interface, using HTTP protocols.
        - Backend performs the necessary operations and returns a response to the Frontend.
    - Database
        - Backend receives a request from the Frontend that requires data retrieval or manipulation, it formulates a query that is sent to the Database.
        - Database processes this query and returns the result to the Backend.

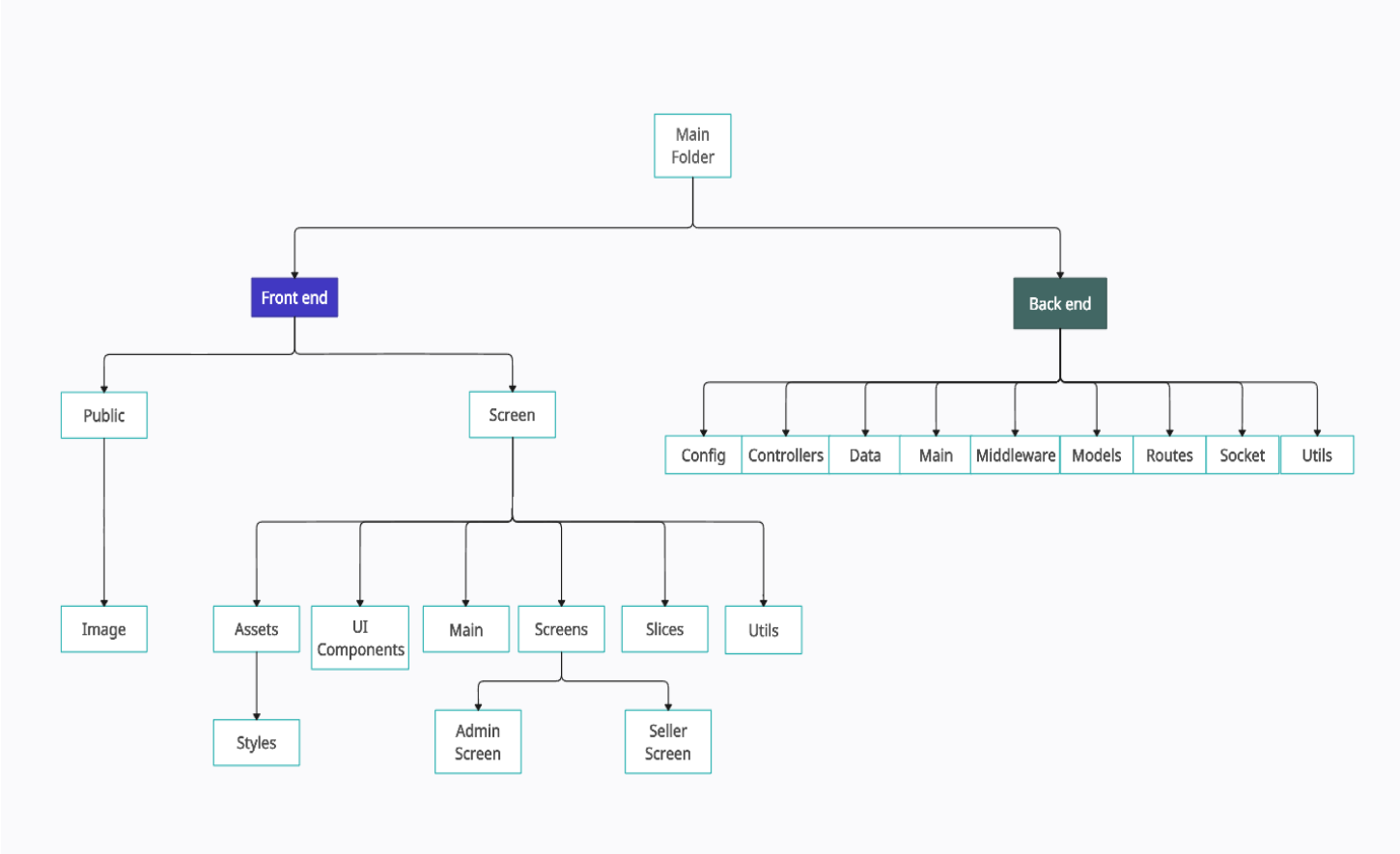
Database

- Description: Database is where all the data for the e-commerce website is stored. The Database receives queries from the Backend, retrieves the necessary data, and sends it back to the Backend.
- Relationship with:
    - Backend
        - Database receives a query from Backend that requires data retrieval or manipulation.
        - Database processes this query and returns the result to the Backend.

VI. Implementation View

Folder Structer:

- This structer is based on



Main Folder

Description:

- This folder contains front end and backend folders which are combined to create a fully functional website.

Front end

Description:

- This folder contains two subfolders: public and screen, containing the source code of the user interface

Public:

Description:

- The public folder contains a subfolder called Images and images for interface screens such as products, logos, ...

Screen:

Description:

- It contains the main parts of the user interface, including components such as widgets (utils), states (slices), image resources (assets), and user interface components (UI components). These parts work together to build and display the main screens of the sales application. The "screen" directory can help organize source code clearly and support efficient project development and maintenance.

Assets:

Description:

- This folder contains subfolder called style which mainly focuses on formatting and decorating elements in the screen part of the application, helping to improve user experience and create an attractive interface.

UI component:

Description:

- The "UI components" folder contains important user interface components. These components include functions such as admin login, chat list display, payment progress steps, product reviews, and more. Components are designed for reuse and increased user interaction. The application becomes flexible, easy to maintain and provides a better online shopping experience for users.

Main:

Description:

- The "main" folder encapsulates the core functionality of a front-end application handling routing, state management, authentication, and integrating various libraries to create a user-friendly web interface.

Screens:

Description:

- This folder contains user cases tailored for specific roles as well as general use cases

Slice:

Description:

- The "slices" folder houses Redux-related files that compartmentalize the application's state management into distinct parts known as "slices." Each file within this folder represents a separate slice responsible for managing specific aspects of the application state, such as authentication, shopping cart operations, chat functionality, orders, products, and user-related data.

Back-end Folder

Description:

- Folder following a model-view-controller (MVC). The separation of concerns into different folders makes the codebase more maintainable and scalable. Check each file and folder's content for a more detailed understanding of the implementation.

Config Folder

Description:

- Contains configuration files for the backend application.
- Commonly includes settings for databases, external services, and environment variables

Controller Folder

Description:

- Holds the controller files responsible for handling incoming HTTP requests.
- Controllers usually process requests, interact with models, and send back responses.

Data Folder

Description:

- The data folder, as a whole, serves the purpose of organizing and centralizing static data used by the application.
- This separation of data from the rest of the application logic allows for easier maintenance, updates, and potential expansion of the dataset.
- It also simplifies testing and development by providing a consistent and predefined dataset that can be used in various scenarios.

Main Folder

Description:

- The main folder is pivotal to your backend application's setup, handling HTTP server creation, WebSocket integration, and database seeding.
- server.js orchestrates the overall server setup, routing, and interaction with the database.
- seeder.js offers a convenient way to initialize or wipe data from the database, useful during development and testing phases.

Middleware Folder

Description:

- Contains middleware components that intercept and process requests before they reach the controllers.
- Middleware is often used for tasks such as authentication, logging, or modifying request/response objects

Model Folder

Description:

- Foundational repository for MongoDB data models implemented with Mongoose in the application.
- Each individual file within this directory corresponds to a distinct entity and serves to meticulously define both the structural composition and the logic governing interactions with the data.

Routes Folder

Description:

- The hub for managing endpoint routing in the backend application.
- Each file contained within this directory is dedicated to a specific category of API routes, defining the way various client requests are managed and processed.

Socket Folder

Description:

- Establishes and configures WebSocket communication using the Socket.io library. It exports a function that takes an HTTP server as a parameter, creating a WebSocket server to handle real-time bidirectional communication.

Utils Folder (Back-end)

Description:

- Within this file, a comprehensive set of utility functions and authorization-related logic is meticulously crafted to empower the backend application.

Image

Description:

- Within this file, a comprehensive set of images to use on the website.

Styles

Description:

- Within this file, a comprehensive set of styles crafted to styling the website.

Admin Screen

Description:

- Within this file, a comprehensive set of screens for user 'Admin', include: OrderListScreen, UserListScreen, UserEditScreen.

Seller Screen

Description:

- Within this file, a comprehensive set of screens for user 'Seller, include: OrderListScreen, ProductEditScreen, ProductListScreen.

Utils (Front-end)

Description:

- Within this file, a comprehensive set of utility functions and authorization-related logic is meticulously crafted to empower the front-end application.