Group02-Shinra Tensei

SOPE
Software Architecture Document

Version 1.11

Revision History

Date	Version	Description	Author
31/07/2024	1.0	Update Use-Case Model, Logical View	Huỳnh Thiên Thuận
01/08/2024	1.1	Update Logical view	Trần Đức Tùng
01/08/2024	1.2	Update Logical View	Huỳnh Thiên Thuận
01/08/2024	1.3	Update Introduction	Lê Trần Kim Oanh
02/08/2024	1.4	Update Architectural Goals and Constraints	Trần Đức Tùng
02/08/2024	1.5	Update component Authentication, Event, Product	Huỳnh Nhật Nam
02/08/2024	1.6	Update Use-Case Model	Lê Trần Kim Oanh
02/08/2024	1.7	Update component Order	Huỳnh Thiên Thuận
02/08/2024	1.8	Update component Withdraw	Lê Trần Kim Oanh
03/08/2024	1.9	Update component Payment, Chat, DiscountCode	Trần TIến Lợi
16/08/2024	1.10	Update Implementation View	Huỳnh Nhật Nam
16/08/2024	1.11	Update Deployment	Huỳnh Thiên Thuận

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

The purpose of this Software Architecture Document is to outline the architectural design of the Sope e-commerce website. It provides a comprehensive description of the system's structure, components, and interactions, ensuring that the solution aligns with business requirements.

1.2 Scope

This document covers all architectural components necessary for building the Sope e-commerce platform, including the front-end, back-end, database.

1.3 Definitions, Acronyms, Abbreviations:

None.

1.4 References:

Vision Document.

1.5 Overview:

This Software Architecture Document outlines the design and structure of the Sope e-commerce platform. It details the architectural goals, use-case models, system components, and their interactions. The document also covers deployment strategies and implementation details, providing a comprehensive guide to ensure the platform meets its business and technical requirements.

2. Architectural Goals and Constraints

2.1 System requirements :

- The server of the website will be operated under windows 11 operating system.
- The website should be compatible with major web browsers, including Chrome, Firefox, Opera, …
- The website can be accessible on multiple types of devices, including desktop computer, tablets, smart phones, …

2.2 Performance Requirements :

- The website should provide fast response times to ensure a smooth user experience. Maximum response time is 10 seconds, maximum screen refreshing time is 5 seconds.
  The website can process at most 200 requests per second.
  Mean time to failure is about a month.
- Time to restart after a failure is about one day.

2.3 Environmental Requirements :

None.

2.4 Quality ranges :

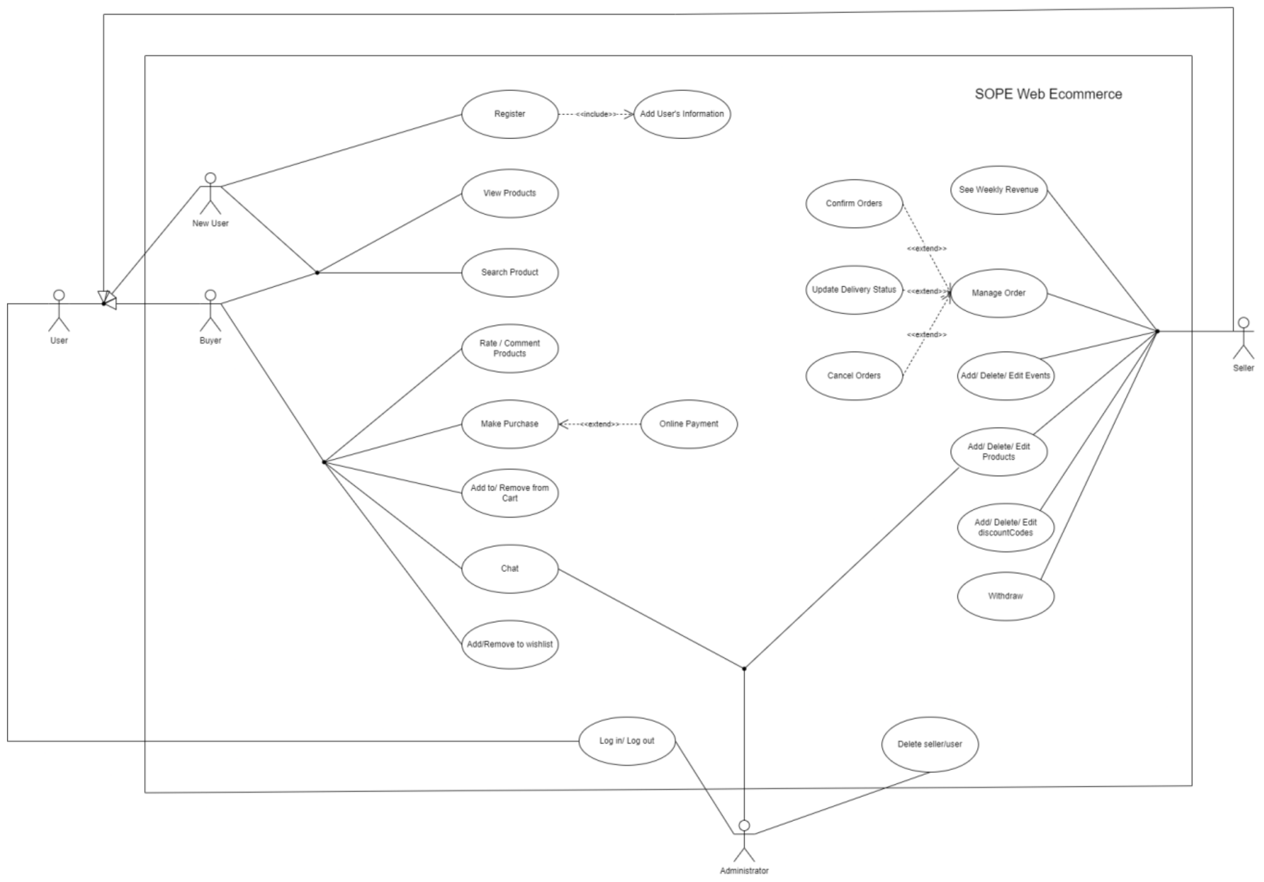
- Availability: The website can be available 23 hours a day, 7 days a week, update every 3 months at least, the downtime will not exceed 1 hour a day.
- Usability: The website shall be easy-to-use and shall be appropriate for a wide range of students in universities, Companies affiliated with the universities.
- Maintainability: The website shall be designed for ease of maintenance by the development team,

data can be modified without restarting the0 server.

2.5 Constraints :

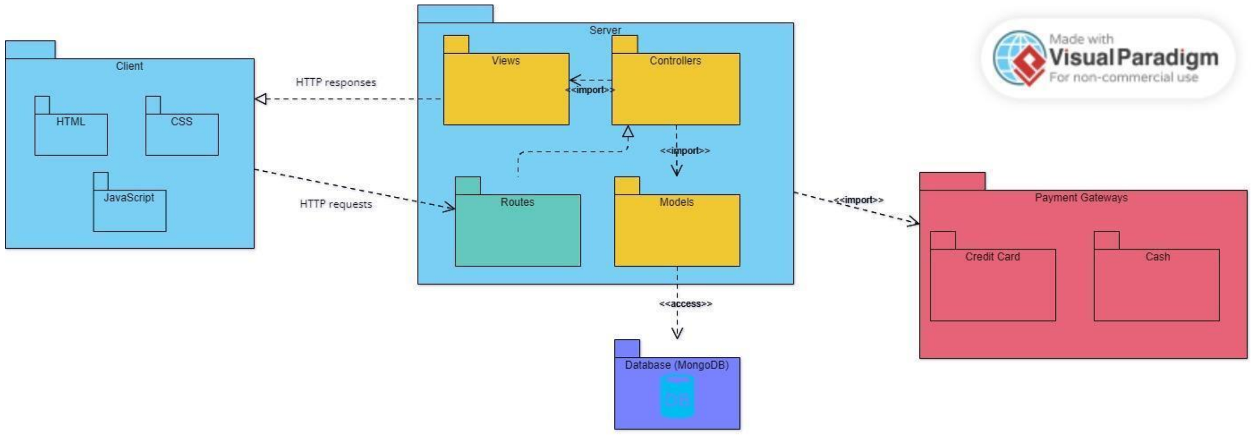
- The system shall not require any hardware development or procurement.
- The number of products is limited to the sellers' supply capacity.

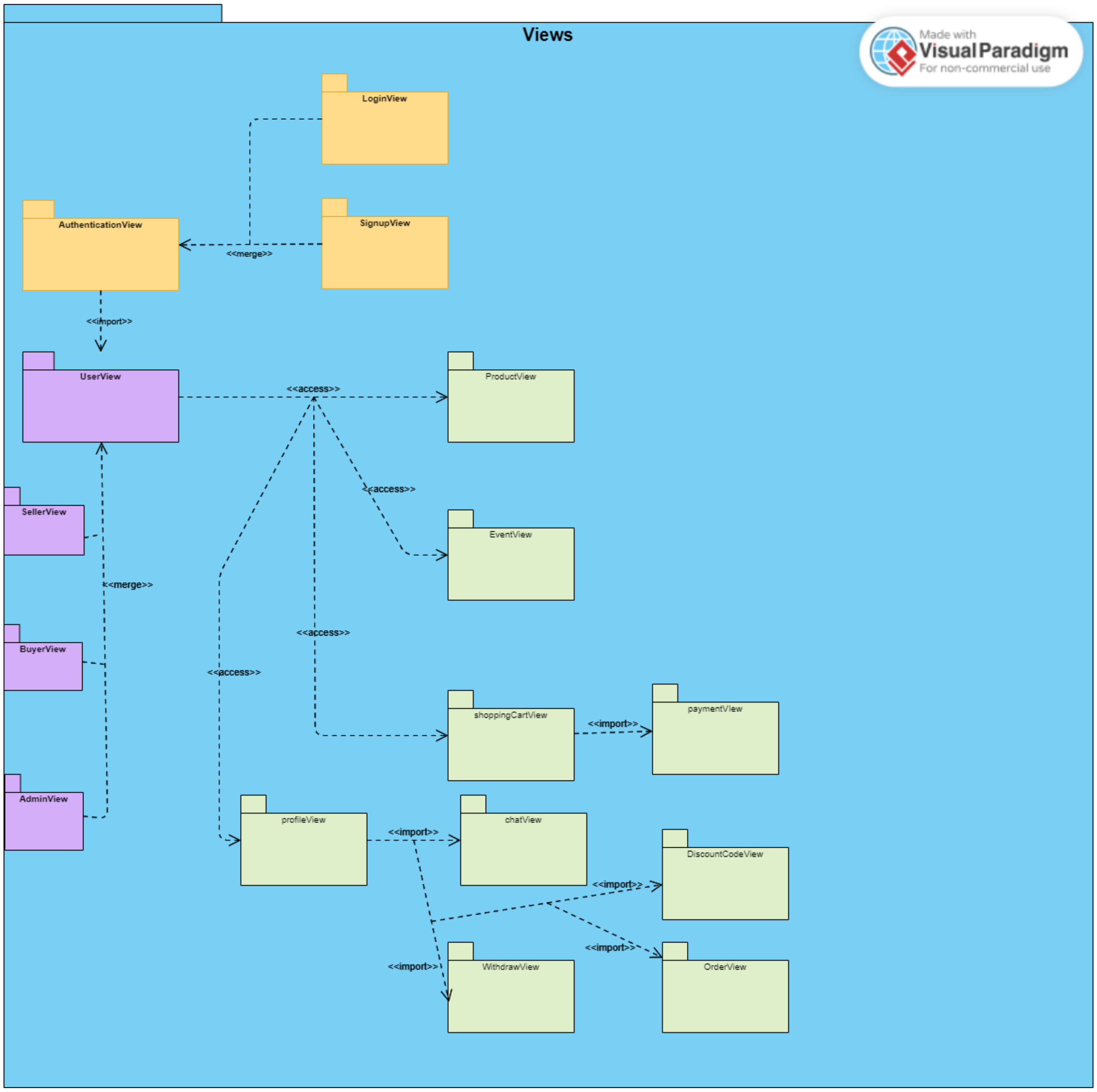
3. Use-Case Model

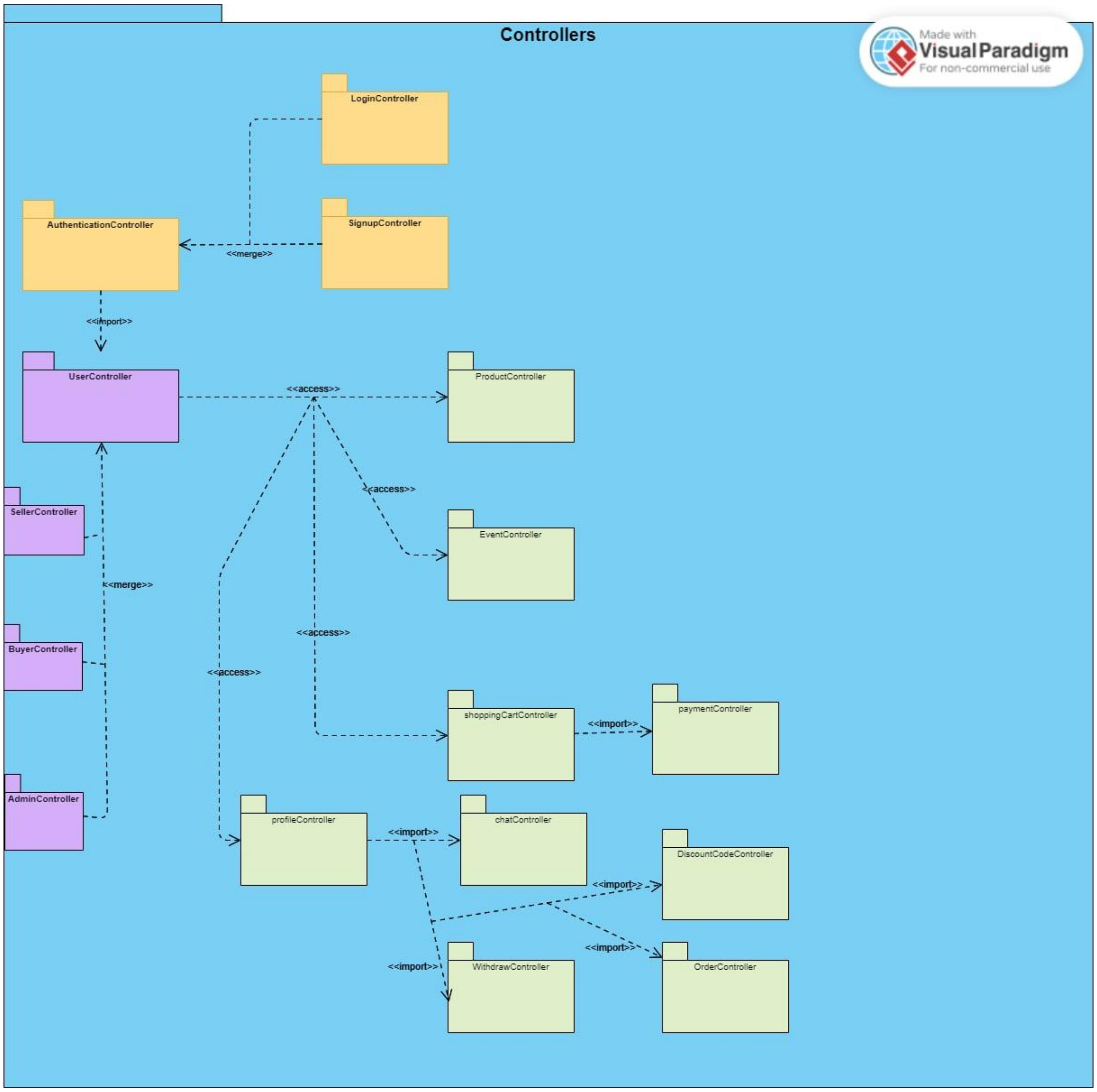


4. Logical View

The logic view package diagram is illustrated as below:

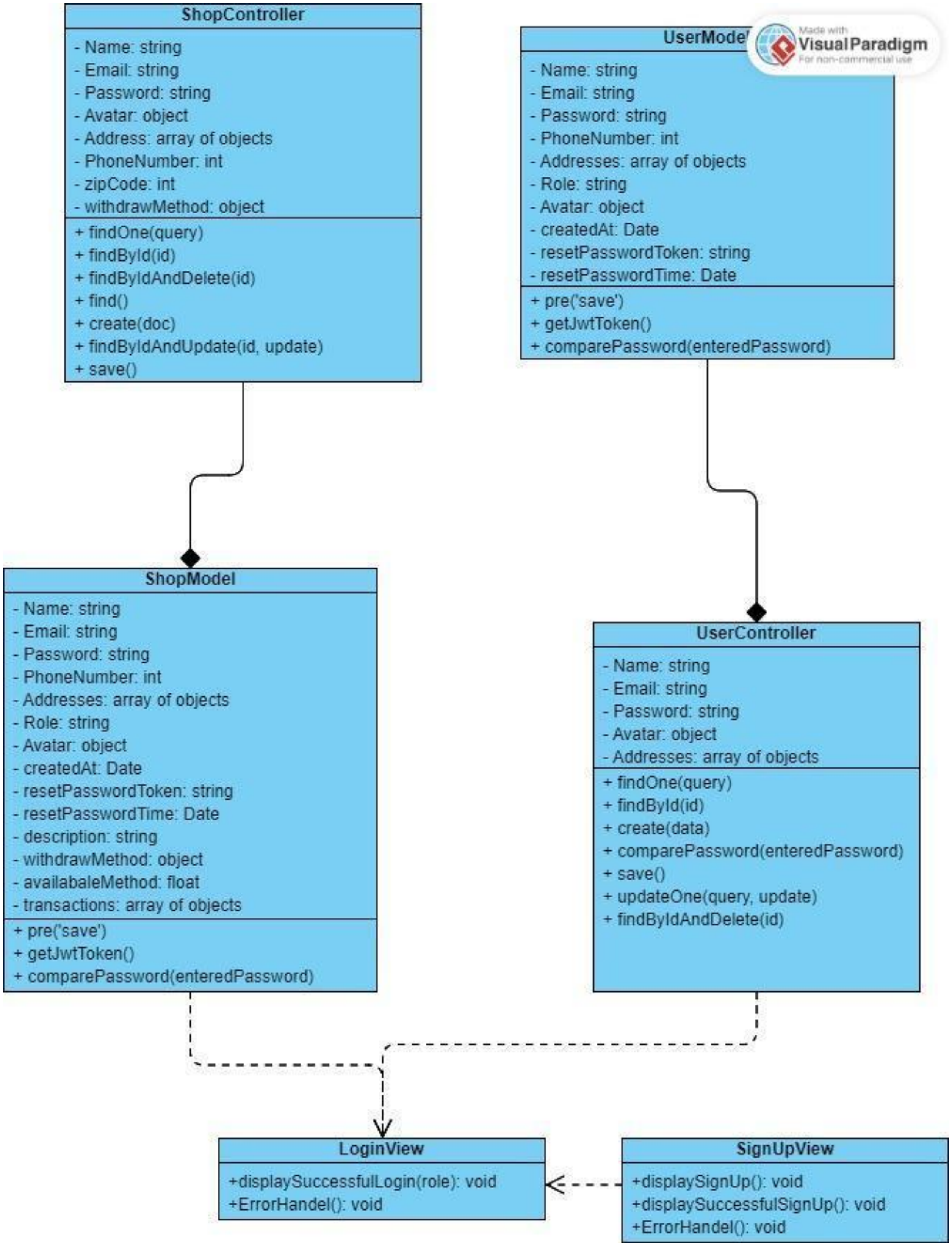

Client
HTML
CSS
JavaScript
HTTP responses
HTTP requests
Server
Views
Controllers
<<import>>
<<import>>
Routes
Models
<<access>>
Database (MongoDB)
<<import>>
Payment Gateways
Credit Card
Cash
Made with Visual Paradigm For non-commercial use


Views
Made with Visual Paradigm For non-commercial use
LoginView
SignupView
AuthenticationView
<<merge>>
<<import>>
UserView
<<access>>
ProductView
<<access>>
EventView
SellerView
<<merge>>
BuyerView
AdminView
<<access>>
<<access>>
shoppingCartView
<<import>>
paymentView
profileView
<<import>>
chatView
DiscountCodeView
<<import>>
<<import>>
WithdrawView
<<import>>
OrderView



4.1 Component: Authentication

4.1.1 Class diagram



ShopController
- Name: string
- Email: string
- Password: string
- Avatar: object
- Address: array of objects
- PhoneNumber: int
- zipCode: int
- withdrawMethod: object
+ findOne(query)
+ findById(id)
+ findByIdAndDelete(id)
+ find()
+ create(doc)
+ findByIdAndUpdate(id, update)
+ save()
UserModel
Made with Visual Paradigm For non-commercial use
- Name: string
- Email: string
- Password: string
- PhoneNumber: int
- Addresses: array of objects
- Role: string
- Avatar: object
- createdAt: Date
- resetPasswordToken: string
- resetPasswordTime: Date
+ pre('save')
+ getJwtToken()
+ comparePassword(enteredPassword)
ShopModel
- Name: string
- Email: string
- Password: string
- PhoneNumber: int
- Addresses: array of objects
- Role: string
- Avatar: object
- createdAt: Date
- resetPasswordToken: string
- resetPasswordTime: Date
- description: string
- withdrawMethod: object
- availabaleMethod: float
- transactions: array of objects
+ pre('save')
+ getJwtToken()
+ comparePassword(enteredPassword)
UserController
- Name: string
- Email: string
- Password: string
- Avatar: object
- Addresses: array of objects
+ findOne(query)
+ findById(id)
+ create(data)
+ comparePassword(enteredPassword)
+ save()
+ updateOne(query, update)
+ findByIdAndDelete(id)
LoginView
+displaySuccessfulLogin(role): void
+ErrorHandel(): void
SignUpView
+displaySignUp(): void
+displaySuccessfulSignUp(): void
+ErrorHandel(): void

4.1.2 LoginView - user interface

- displaySuccessfulLogin(role): displays a successful login message to the user based on their role.
- ErrorHandler(): handles login-related errors.

4.1.3 SignUpView - user interface

- displaySignUp(): displays the sign-up form to the user.
- displaySuccessfulSignUp(): displays a successful sign-up message to the user.
- ErrorHandler(): handles sign-up-related errors.

4.2.4 ShopController - handles the business logic for processing shop-related requests

- findOne(query): finds one shop based on the query.
- findById(id): finds a shop by its ID.
- findByIdAndDelete(id): finds a shop by its ID and deletes it.
- find(): retrieves all shops.
- create(doc): creates a new shop and stores it in the database.
- findByIdAndUpdate(id, update): finds a shop by its ID and updates it.
- save(): saves the shop information to the database.

4.2.5 UserController - handles the business logic for processing user-related requests

- findOne(query): finds one user based on the query.
- findById(id): finds a user by their ID.
- create(data): creates a new user and stores it in the database.
- comparePassword(enteredPassword): compares the entered password with the stored password.
- save(): saves the user information to the database.
- updateOne(query, update): updates user information based on the query.
- findByIdAndDelete(id): finds a user by their ID and deletes it.

4.2.6 ShopModel - represents the data structure of a shop

- name: string representing the name of the shop.
- email: string containing the shop's email.
- password: string containing the shop's password.
- phoneNumber: int representing the shop's phone number.
- addresses: array of objects containing the shop's addresses.
- zipCode: int representing the shop's zip code.
- avatar: object containing the shop's avatar.
- role: string representing the shop's role.
- createdAt: date when the shop was created.
- resetPasswordToken: string used for resetting the password.
- resetPasswordTime: date when the password was reset.
- description: string containing the shop's description.
- withdrawMethod: object containing the shop's withdrawal method.
- availableMethod: float representing the available method.
- transactions: array of objects containing the shop's transactions.
- pre(save): function to be called before saving the shop information.
- getJwtToken(): generates a JWT token for the shop.
- comparePassword(enteredPassword): compares the entered password with the stored password.

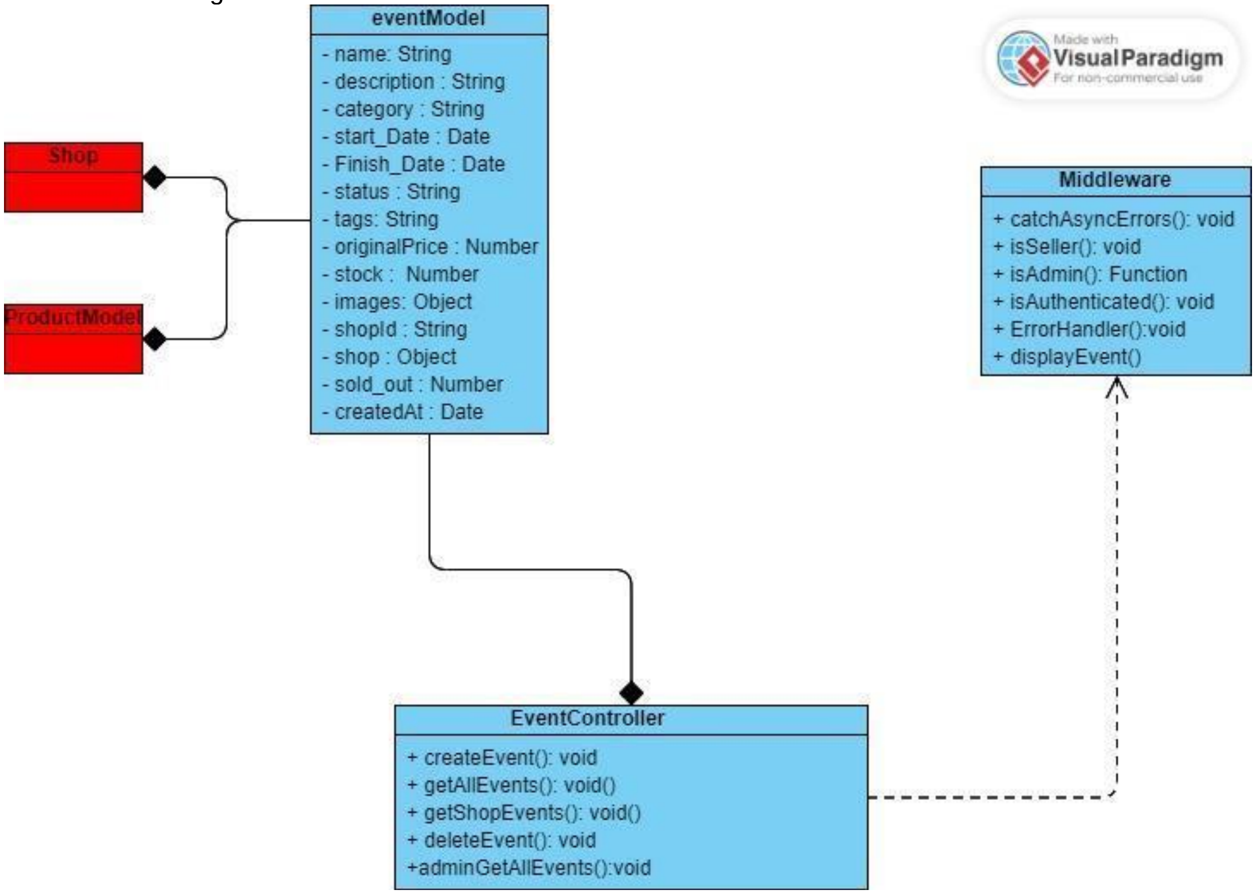
4.2.7 UserModel - represents the data structure of a user

- name: string representing the name of the user.
- email: string containing the user's email.

- password: string containing the user's password.
- phoneNumber: int representing the user's phone number.
- addresses: array of objects containing the user's addresses.
- role: string representing the user's role.
- avatar: object containing the user's avatar.
- createdAt: date when the user was created.
- resetPasswordToken: string used for resetting the password.
- resetPasswordTime: date when the password was reset.
- pre(save): function to be called before saving the user information.
- getJwtToken(): generates a JWT token for the user.
- comparePassword(enteredPassword): compares the entered password with the stored password.

4.2 Component: Event

4.2.1 Class diagram



4.2.2 EventView - user interface

- displayEvents(): displays a list of events to the user.

4.2.3 EventController - handles the business logic for processing event-related requests

- createEvent(): creates a new event and stores it in the database.
- getAllEvents(): retrieves all events for a user or seller.
- getShopEvents(): retrieves events related to a specific shop.
- deleteEvent(): deletes an existing event.

- adminGetAllEvents(): retrieves all events for an admin.

4.2.4 EventModel - represents the data structure of an event

- name: string representing the name of the event.
- description: string containing the event description.
- category: string representing the event category.
- startDate: date when the event starts.
- finishDate: date when the event finishes.
- status: string representing the event status.
- tags: string containing tags related to the event.
- originalPrice: number representing the original price of the event.
- stock: number representing the stock available for the event.
- images: objects containing images related to the event.
- shopId: string representing the ID of the shop hosting the event.
- shop: object containing shop details.
- soldOut: number representing the number of sold out tickets/items.
- createdAt: date when the event was created.

4.2.5 Middleware - provides supporting functionalities for the application

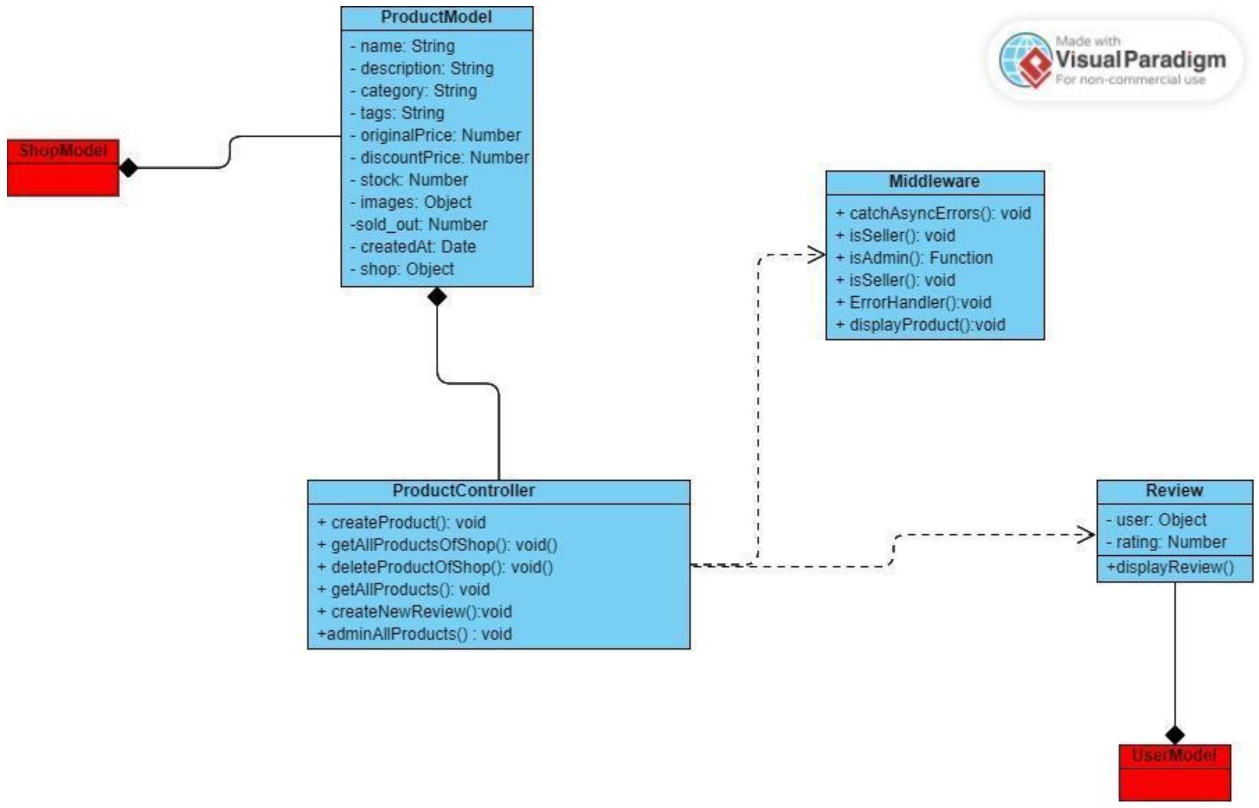
- catchAsyncErrors(): handles asynchronous errors.
- isSeller(): checks if the user is a seller.
- isAdmin(): checks if the user is an admin.
- isAuthenticated(): checks if the user is authenticated.
- errorHandler(): handles application errors.

4.2.6 Other

- The other class takes the supporting roles for the key classes (Model, View, Controller) in the current software component.
- The class ProductModel will be defined in other components in our software architecture.

4.3 Component: Product

4.3.1 Class diagram



4.3.2 ProductView - user interface
- displayProducts(): displays a list of products to the user.

4.3.3 ProductController - handles the business logic for processing product-related requests
- createProduct(): creates a new product and stores it in the database.
- getAllProductsOfShop(): retrieves all products for a specific shop.
- deleteProductOfShop(): deletes an existing product from a shop.
- getAllProducts(): retrieves all products.
- createNewReview(): creates a new review for a product.
- adminAllProducts(): retrieves all products for an admin.

4.2.4 ProductModel - represents the data structure of a product
- name: string representing the name of the product.
- description: string containing the product description.
- category: string representing the product category.
- tags: string containing tags related to the product.
- originalPrice: number representing the original price of the product.
- discountPrice: number representing the discounted price of the product.
- stock: number representing the stock available for the product.
- images: objects containing images related to the product.
- soldOut: number representing the number of sold-out items.
- createdAt: date when the product was created.
- shop: object containing shop details.

4.3.5 Middleware - provides supporting functionalities for the application

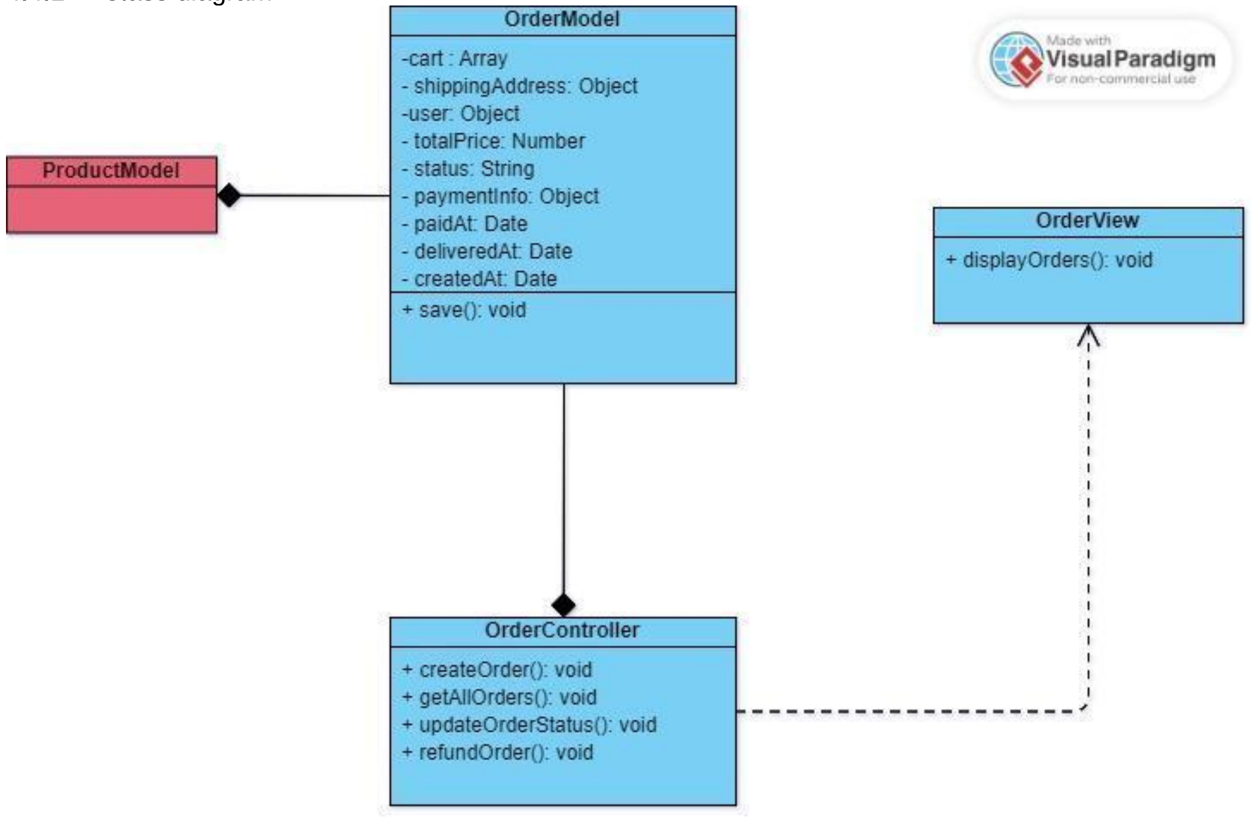
- catchAsyncErrors(): handles asynchronous errors.
- isSeller(): checks if the user is a seller.
- isAdmin(): checks if the user is an admin.
- errorHandler(): handles application errors.
- displayProduct(): displays product details.

4.3.6 Review - represents the data structure of a product review
- user: object containing user information.
- rating: number representing the review rating.
- displayReview(): displays the review details.

4.3.7 Other
- The other class takes the supporting roles for the key classes (Model, View, Controller) in the current software component.
- The class ShopModel will be defined in other components in our software architecture.
- The class UserModel will be defined in other components in our software architecture.

4.4 Component: Order

4.4.1 Class diagram



4.4.2 OrderView - user interface
- displayOrders(): displays a list of orders to the user.

4.4.3 OrderController - handles the business logic for processing order-related requests
- createOrder(): creates a new order and stores it in the database.

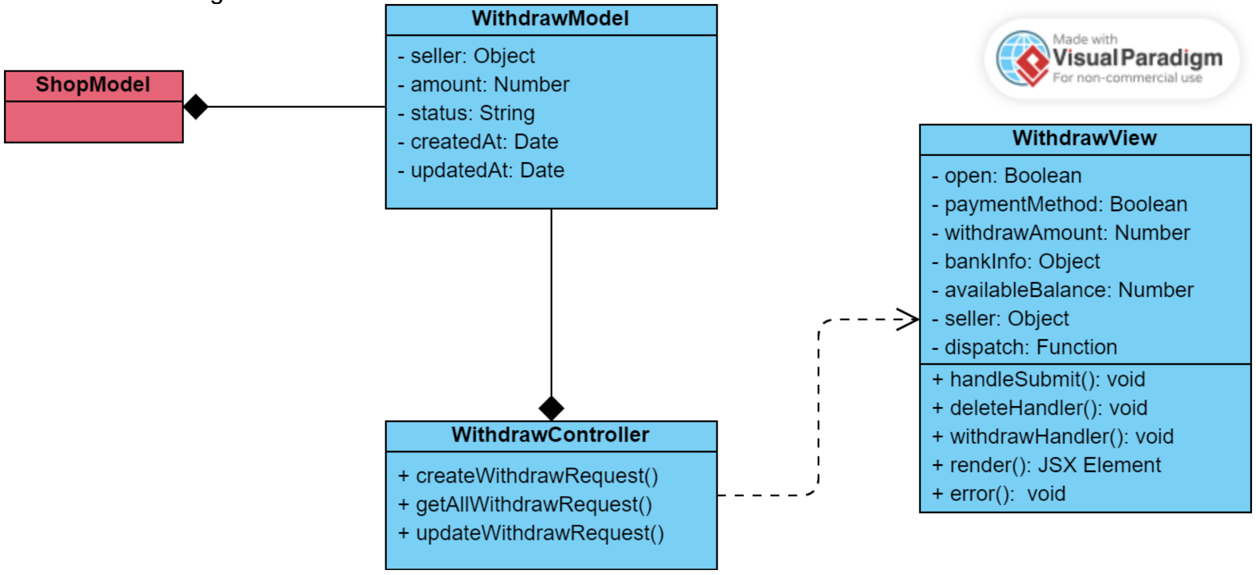
- getAllOrders(): retrieves all orders for a user or seller.
- updateOrderStatus(): updates the status of an existing order.
- refundOrder(): processes a refund for an order.

4.4.4 OrderModel - represents the data structure of an order
- cart: array containing items in the cart.
- shippingAddress: object containing the shipping address.
- user: object containing user information.
- totalPrice: total price of the order.
- status: status of the order (e.g., "Processing", "Delivered").
- paymentInfo: object containing payment details.
- paidAt: date when the order was paid.
- deliveredAt: date when the order was delivered.
- createdAt: date when the order was created.
- save(): save the order information to the database.

4.4.5 Other
- The other class takes the supporting roles for the key classes (Model, View, Controller) in the current software component.
- The class ProductModel will be defined in other components in our software architecture.

4.5 Component: Withdraw

4.5.1 Class diagram



4.5.2 WithdrawView - user interface
- open: Boolean: Controls the visibility of the withdrawal modal.
- paymentMethod: Boolean: Indicates whether the payment method form is being displayed.
- withdrawAmount: Number: Stores the amount the seller wants to withdraw. Initialized to 50.
- bankInfo: Object: Contains bank-related information for withdrawal, including:
  - bankName: String
  - bankCountry: String
  - bankSwiftCode: String
  - bankAccountNumber: String

- bankHolderName: String
- bankAddress: String

- availableBalance: Number: Contains the seller's available balance.
- seller: Object: Contains the seller's data from the Redux store.
- dispatch: Function: Redux dispatch function for sending actions to the store.
- handleSubmit(): Handles the submission of the new withdrawal method form.
- deleteHandler(): Deletes the current withdrawal method.
- error(): Displays an error message for insufficient balance.
- withdrawHandler(): Processes the withdrawal request.
- render(): Renders the UI for managing withdrawals.
- error(): Displays an error message for insufficient balance.

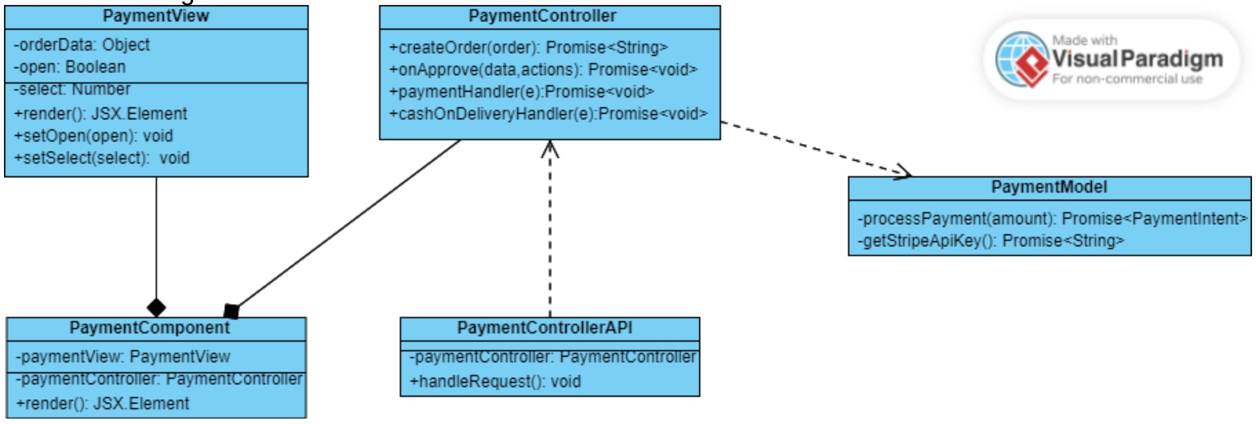
4.5.3 WithdrawController - handles the business logic for processing withdraw-related requests
- createWithdrawRequest(): Creates a new withdrawal request and updates the seller's balance.
- getAllWithdrawRequest(): Retrieves and returns all withdrawal requests.
- updateWithdrawRequest(): Updates the status of a withdrawal request and notifies the seller.

4.5.4 WithdrawModel - represents the data structure of withdraw
- seller: Object: Stores the seller's information for the withdrawal request.
- amount: Number: Represents the requested withdrawal amount.
- status: String: Indicates the current status of the withdrawal, defaulting to "Processing".
- createdAt: Date: Timestamp of when the withdrawal request was created.
- updatedAt: Date: Timestamp of the last update to the withdrawal request.

4.5.5 Other
- The other class takes the supporting roles for the key classes (Model, View, Controller) in the current software component.
- The class ShopModel will be defined in other components in our software architecture.

4.6 Component: Payment

4.6.1 Class diagram


PaymentView
-orderData: Object
-open: Boolean
-select: Number
+render(): JSX.Element
+setOpen(open): void
+setSelect(select): void
PaymentController
+createOrder(order): Promise<String>
+onApprove(data,actions): Promise<void>
+paymentHandler(e):Promise<void>
+cashOnDeliveryHandler(e):Promise<void>
Made with Visual Paradigm For non-commercial use
PaymentModel
-processPayment(amount): Promise<PaymentIntent>
-getStripeApiKey(): Promise<String>
PaymentComponent
-paymentView: PaymentView
-paymentController: PaymentController
+render(): JSX.Element
PaymentControllerAPI
-paymentController: PaymentController
+handleRequest(): void

4.6.2 PaymentView:
- orderData: Object: This attribute holds the data related to the order that needs to be processed for payment.
- open: Boolean: A boolean flag that indicates whether the payment view is currently open or not.
- select: Number: This attribute keeps track of the selected payment option.

- render(): JSX.Element: This method is responsible for rendering the payment view UI as a JSX element. It includes the logic to display the order data and payment options to the user.
- setOpen(open: Boolean): void: This method is used to set the open attribute. It opens or closes the payment view based on the boolean value passed to it.
- setSelect(select: Number): void: This method is used to update the select attribute with the chosen payment option.

4.6.3 PaymentComponent:
- paymentView: PaymentView: An instance of the PaymentView class that manages the UI related to payments.
- paymentController: PaymentController: An instance of the PaymentController class that handles the business logic for processing payments.
- render(): JSX.Element: This method renders the PaymentComponent, integrating both the PaymentView and the PaymentController to provide a seamless payment experience to the user.

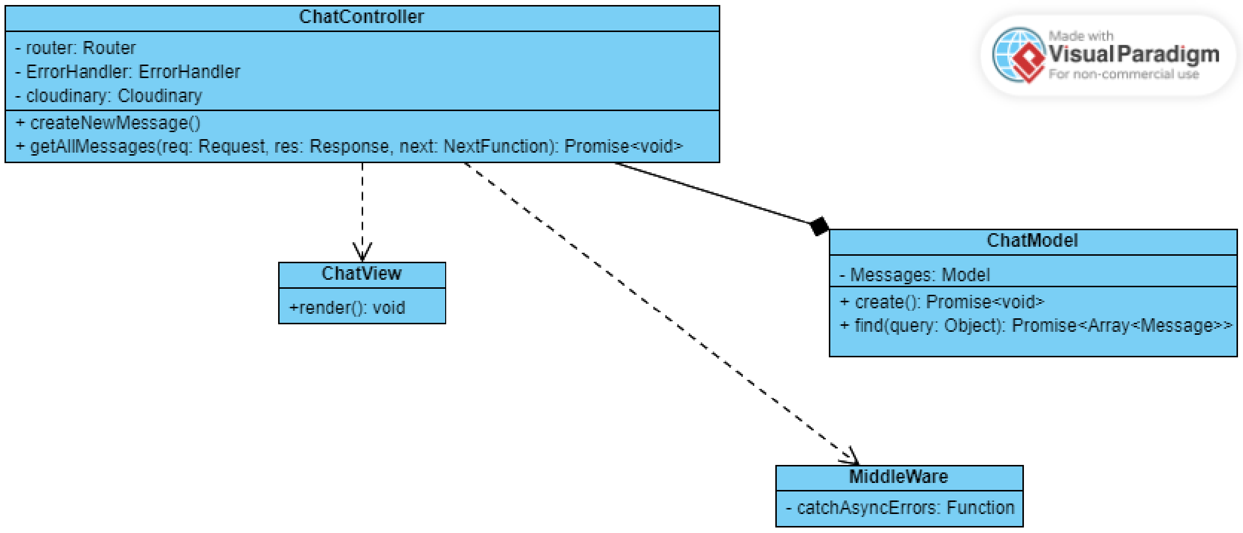
4.6.4 PaymentController
- createOrder(order: Object): Promise<String>: This method creates a new order using the given order data and returns a promise that resolves to a string, typically an order ID.
- onApprove(data: Object, actions: Object): Promise<void>: This method handles the approval process once the payment is authorized. It takes the payment data and actions as parameters and returns a promise that resolves when the approval process is complete.
- paymentHandler(e: Event): Promise<void>: This method handles payment-related events, processing the payment based on the event data. It returns a promise that resolves when the payment is processed.
- cashOnDeliveryHandler(e: Event): Promise<void>: This method specifically handles cash-on-delivery payment events. It processes the cash-on-delivery payment and returns a promise that resolves when the processing is complete.

4.6.5 PaymentModel
- processPayment(amount: Number): Promise<PaymentIntent>: This method processes a payment for the given amount. It interacts with the payment gateway to create a payment intent and returns a promise that resolves to a PaymentIntent object.
- getStripeApiKey(): Promise<String>: This method retrieves the Stripe API key required for processing payments. It returns a promise that resolves to the API key as a string.

4.6.6 PaymentControllerAPI
- paymentController: PaymentController: An instance of the PaymentController class that handles the business logic for payments.
- handleRequest(): void: This method handles incoming API requests related to payment processing. It delegates the requests to the PaymentController for further processing.

4.7 Component: Chat

4.7.1 Class diagram


ChatController
- router: Router
- ErrorHandler: ErrorHandler
- cloudinary: Cloudinary
+ createNewMessage()
+ getAllMessages(req: Request, res: Response, next: NextFunction): Promise<void>
ChatView
+render(): void
ChatModel
- Messages: Model
+ create(): Promise<void>
+ find(query: Object): Promise<Array<Message>>
MiddleWare
- catchAsyncErrors: Function
Made with Visual Paradigm For non-commercial use

4.7.2 ChatModel

- Messages: Model: A database model that represents the chat messages schema.
- create(): Promise<void>: This method creates a new chat message record in the database. It returns a promise that resolves when the message is successfully saved.
- find(query: Object): Promise<Array<Message>>: This method retrieves chat messages from the database based on the provided query. It returns a promise that resolves to an array of Message objects that match the query criteria.

4.7.3 ChatView

- render(): void: This method renders the chat view, displaying chat messages and other related UI elements to the user. It includes the logic for presenting the chat interface based on the current state and data.

4.7.4 ChatController

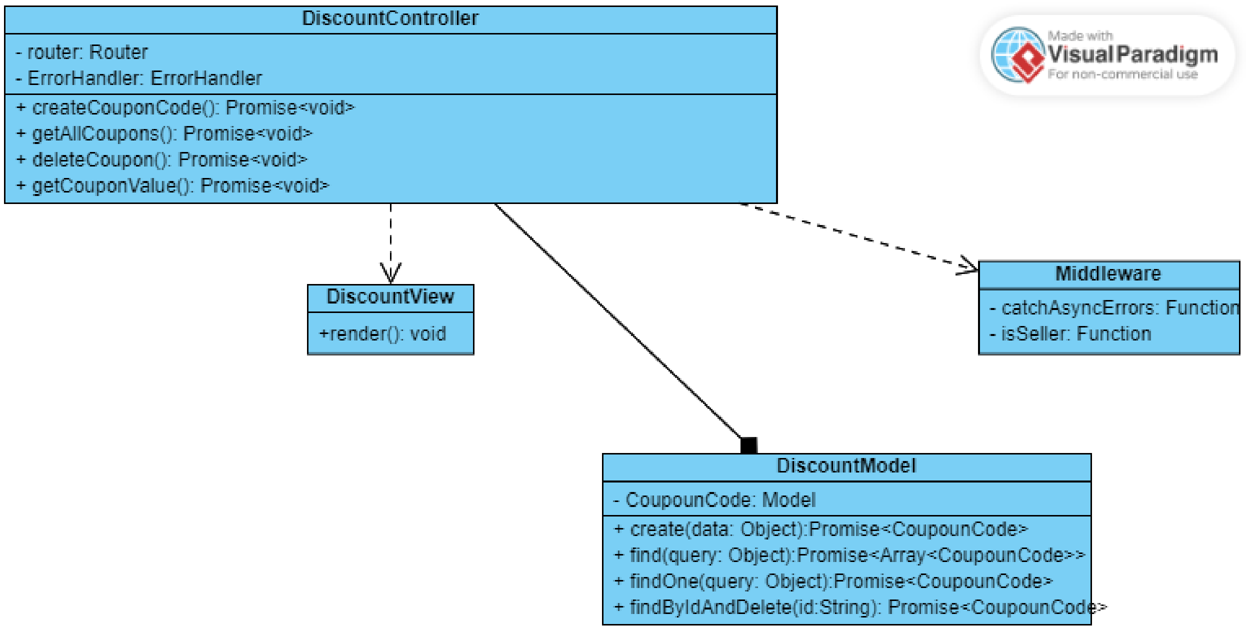
- router: Router: An instance of a Router that defines the routes for the chat-related operations.
- ErrorHandler: ErrorHandler: An instance of an ErrorHandler that manages error responses for chat operations.
- cloudinary: Cloudinary: An instance of Cloudinary used for managing media storage and retrieval in the cloud.

4.7.5 MiddleWare

- catchAsyncErrors: Function: A function that wraps asynchronous operations to catch and handle errors. It ensures that any errors occurring in asynchronous code are properly propagated to the ErrorHandler.

4.8 Component: DiscountCode

4.8.1 Class diagram


DiscountController
- router: Router
- ErrorHandler: ErrorHandler
+ createCouponCode(): Promise<void>
+ getAllCoupons(): Promise<void>
+ deleteCoupon(): Promise<void>
+ getCouponValue(): Promise<void>
Made with Visual Paradigm For non-commercial use
DiscountView
+render(): void
Middleware
- catchAsyncErrors: Function
- isSeller: Function
DiscountModel
- CoupounCode: Model
+ create(data: Object):Promise<CoupounCode>
+ find(query: Object):Promise<Array<CoupounCode>>
+ findOne(query: Object):Promise<CoupounCode>
+ findByIdAndDelete(id:String): Promise<CoupounCode>

4.8.2 DiscountModel

- CouponCode: Model: A database model that represents the schema for coupon codes.
- create(data: Object): Promise<CouponCode>: This method creates a new coupon code record in the database using the provided data and returns a promise that resolves to the created CouponCode object.
- find(query: Object): Promise<Array<CouponCode>>: This method retrieves an array of coupon codes that match the provided query criteria and returns a promise that resolves to an array of CoupounCode objects.
- findOne(query: Object): Promise<CouponCode>: This method retrieves a single coupon code that matches the provided query criteria and returns a promise that resolves to a CoupounCode object.
- findByIdAndDelete(id: String): Promise<CouponCode>: This method deletes a coupon code by its ID and returns a promise that resolves to the deleted CouponCode object.

4.8.3 DiscountView

- render(): void: This method renders the discount view, displaying discount information, coupon codes, and other related UI elements to the user. It includes the logic for presenting the discount interface based on the current state and data.

4.8.4 DiscountController

- router: Router: An instance of a Router that defines the routes for discount-related operations.
- ErrorHandler: ErrorHandler: An instance of an ErrorHandler that manages error responses for discount operations.
- createCouponCode(): Promise<void>: This method handles the creation of a new coupon code. It processes the incoming request, interacts with the DiscountModel to save the coupon code, and manages any associated responses or errors.
- getAllCoupons(): Promise<void>: This method retrieves all coupon codes. It interacts with the DiscountModel to fetch the coupon codes and returns them to the requester.
- deleteCoupon(): Promise<void>: This method handles the deletion of a coupon code. It processes the incoming request, interacts with the DiscountModel to delete the coupon code by

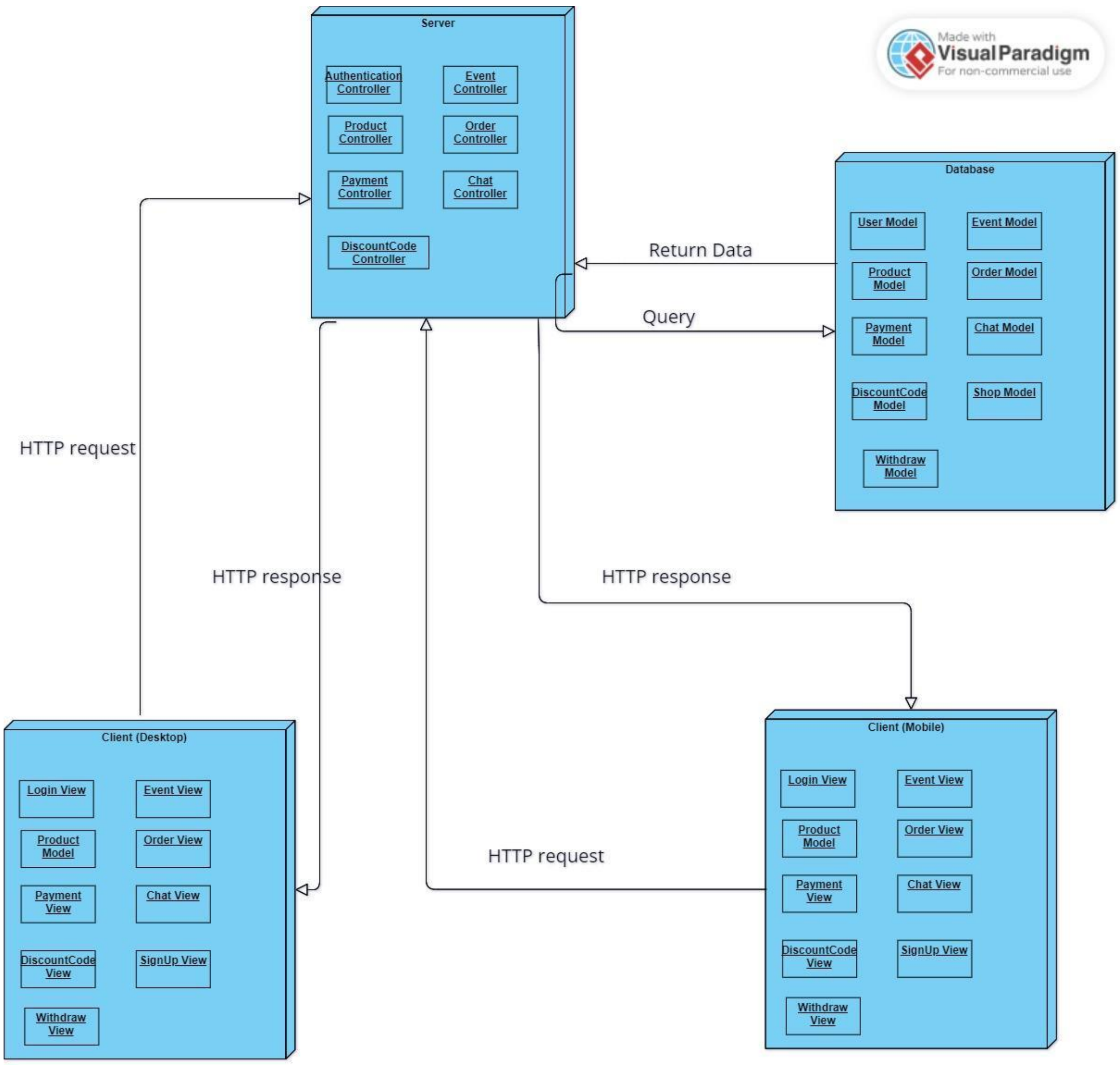
ID, and manages any associated responses or errors.

- getCouponValue(): Promise<void>: This method retrieves the value of a specific coupon code. It interacts with the DiscountModel to fetch the coupon code based on provided criteria and returns the value to the requester.

4.8.5 Middleware

- catchAsyncErrors: Function: A function that wraps asynchronous operations to catch and handle errors. It ensures that any errors occurring in asynchronous code are properly propagated to the ErrorHandler.
- isSeller: Function: A function that checks if the current user has seller permissions. It ensures that certain operations are restricted to users with the appropriate access level.

5. Deployment



Server Node:

-Description: The Web Server is responsible for hosting the web application. It handles HTTP requests from clients, serves static content, and routes requests to the appropriate application components.

-Components:

+Authentication Controller
+Event Controller
+Product Controller
+Order Controller
+Payment Controller
+Chat Controller
+DiscountCode Controller

Database Node:

-Description: The Database is responsible for storing and managing all persistent data for the application. It includes databases for user information, product details, orders, events, payments,...

-Components:

+User Model
+Shop Model
+Event Model
+Product Model
+Order Model
+Withdraw Model
+Payment Model
+Discount Model
+Chat Model

Client (Desktop) Node:

-Description: The Client Node represents the user's device accessing the eCommerce platform via a web browser. It runs the user interface and interacts with the Web Server to perform various operations like browsing products, placing orders, and managing accounts.

-Components:

+Login View
+SignUp View
+Event View
+Product View
+Order View
+Withdraw View
+Payment View
+Chat View
+Discount View

Client (Mobile) Node:

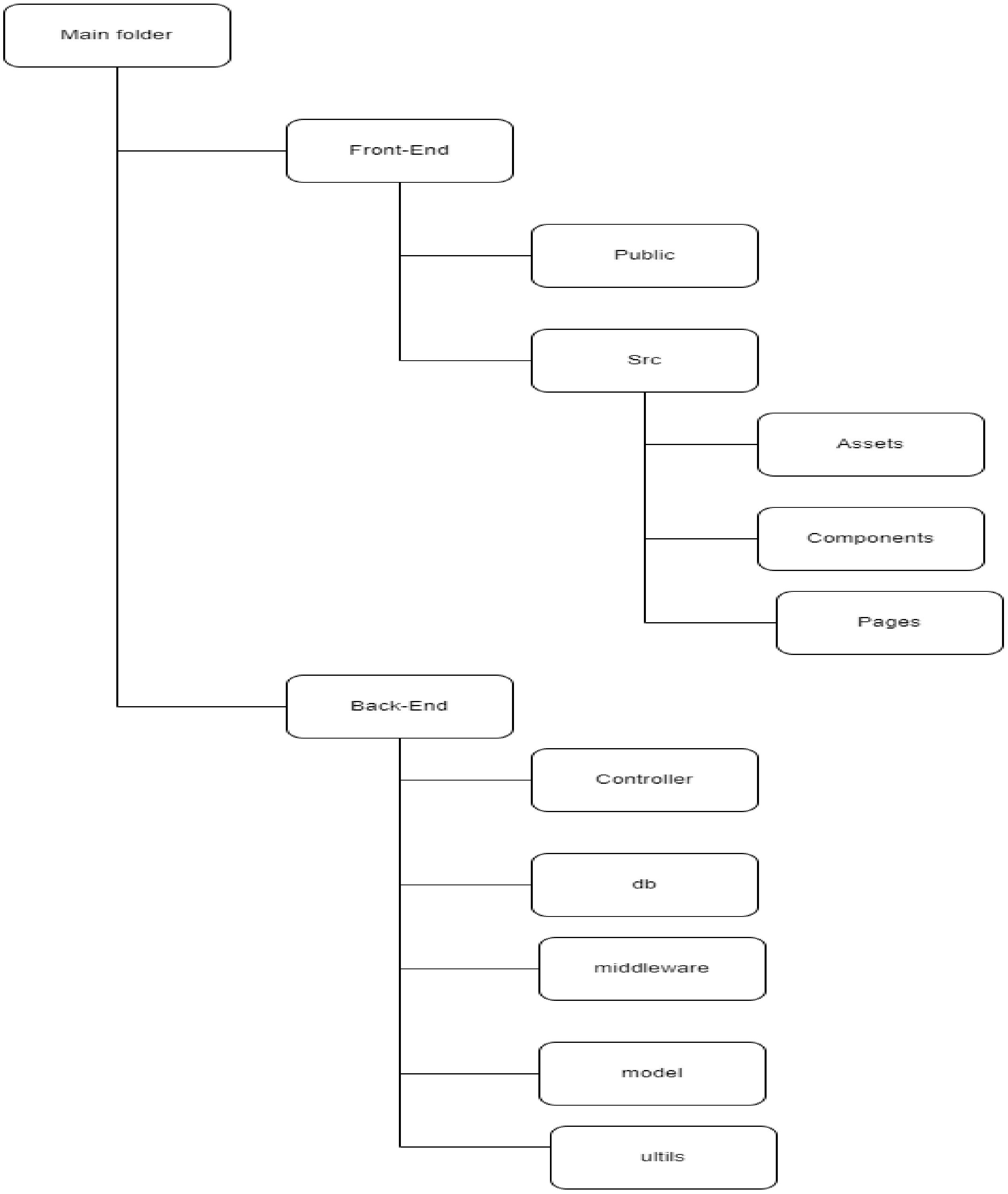
-Description: The Mobile Client Node represents the user's mobile device accessing the eCommerce platform via a mobile application. Similar to the Web Client, it interacts with the Web Server to perform various operations.

-Components:

+Login View
+SignUp View
+Event View

+Product View
+Order View
+Withdraw View
+Payment View
+Chat View
+Discount View

6. Implementation View



Folder Name	File Name
Front-End	
components	UserOrderDetails.jsx
Admin	AdminDashboardMain.jsx
	AllSellers.jsx
	AllWithdraw.jsx
	AllUsers.jsx
	AllProducts.jsx
	AllEvents.jsx
cart	Cart.jsx
Checkout	Checkout.jsx
	CheckoutSteps.jsx
Events	EventCard.jsx
	CountDown.jsx
	Events.jsx
Layout	Header.jsx
	Footer.jsx
	Navbar.jsx
	AdminHeader.jsx
	DropDown.jsx

	Loader.jsx
Login	Login.jsx
Payment	Payment.jsx
Products	ProductDetails.jsx
	SuggestedProduct.jsx
	Ratings.jsx
Profile	ProfileContent.jsx
	ProfileSidebar.jsx
	TrackOrder.jsx
Shop	WithdrawMoney.jsx
	DashboardMessages.jsx
	CreateEvent.jsx
	AllCoupons.jsx
	ShopCreate.jsx
	OrderDetails.jsx
	ShopSettings.jsx
	ShopLogin.jsx
	DashboardHero.jsx
	ShopProfileData.jsx

	ShopInfo.jsx
	AllEvents.jsx
	AllProducts.jsx
	AllRefundOrders.jsx
	AllOrders.jsx
Signup	Signup.jsx
Wishlist	Wishlist.jsx
Page	UserInbox.jsx
	FAQPage.jsx
	AdminDashboardOrders.jsx
	ProductsPage.jsx
	ProductDetailsPage.jsx
	BestSellingPage.jsx
	SellerActivationPage.jsx
	ActivationPage.jsx
	ProfilePage.jsx
	OrderSuccessPage.jsx
	HomePage.jsx
	AdminDashboardPage.jsx

	AdminDashboardWithdraw.jsx
	AdminDashboardProducts.jsx
	AdminDashboardSellers.jsx
	EventsPage.jsx
	AdminDashboardEvents.jsx
	AdminDashboardUsers.jsx
	PaymentPage.jsx
	ShopLoginPage.jsx
	ShopCreate.jsx
	CheckoutPage.jsx
	Login.jsx
	OrderDetailsPage.jsx
	SignupPage.jsx
	TrackOrderPage.jsx
	ShopPreviewPage.jsx
	ShopHomePage.jsx
	ShopAllOrders.jsx
	ShopCreateProduct.jsx

	ShopAllProducts.jsx
	ShopAllCoupouns.jsx
	ShopAllEvents.jsx
	ShopSettingsPage.jsx
	ShopDashboardPage.jsx
	ShopCreateEvents.jsx
	ShopWithDrawMoneyPage.jsx
	ShopInboxPage.jsx
	ShopOrderDetails.jsx
Back-End	
app	app.js
server	server.js
controller	user.js
	shop.js
	order.js
	product.js
	event.js
	withdraw.js

	conversation.js
	coupounCode.js
	message.js
	payment.js
db	Database.js
middleware	auth.js
	error.js
	catchAsyncErrors.js
model	shop.js
	user.js
	order.js
	product.js
	event.js
	withdraw.js
	messages..js
	coupounCode.js
	conversation.js
utils	sendMail.js
	shopToken.js

	jwtToken.js
	ErrorHandler.js