MINISTRY OF EDUCATION AND TRAINING

FPT UNIVERSITY

DETAILED DESIGN DOCUMENT

SWD392 - SE1880

Group 3	
Group Member	Dương Minh Quyền
	Lê Tiến Bình
	Chu Tuấn Ngọc
	Bùi Hoàng Việt
	Nguyễn Tiến Dũng
Supervisor	SangNV
Ext. Supervisor	
Project Code	CSMS

- Hanoi, 10/2025 -

Table of Contents

1 DETAILED DESIGN

1.1 Common Design

1.1.1 Front-End

Page Structure:

Header: Displays the store logo, logged in user information and a logout button. Fixed header, height 70px, dark brown background (#3E2723).

Content: Main area to display dynamic content (data tables, forms, charts, object details,...). Has a margin (margin : 0 auto) to avoid being covered by the header, has width = 100%, content is left aligned

SideBar: To be the left side. contains menu management items, order, ingredient, finance, report. margin: 0 auto, style is Col{3} in react-bootstrap, padding: 0. Main color: #3E2723, back-ground: white

Footer: Displays copyright information ("© 2025 Coffee Management System") and system version. Fixed bottom, height 50px, dark brown background (#3E2723).

Interface and formatting:

Framework/UI Library: Using React with React-Bootstrap.

Fonts: "Roboto", "Open Sans", sans-serif

Font-size:

Title: 18px

Label: 15px

Body: 14px

Main color:

Primary background color: white

Primary color: #B17d74

Error color: #E53935

Success color: #43A047

Spacing:

Margin between elements: m-2

Padding between elements: m-2

Input Components (Input, Label, Button)

Label: Left alignment, lowercase, with : at the end, color #333.

Input: Slightly rounded corners (border-radius: 6px), border color #ccc. When focused: border changes to main color #795548. Placeholder is light gray #9E9E9E.

Button:

Standard size has height: 40px, padding: 3px.

Primary Button has background #795548, white text, hover darker (#5D4037).

Secondary Button: white background, brown border (#795548), brown text.

Table:

Header: background color #D7CCC8, bold (font-weight: 600).

Body: alternating line background color (odd – white, even – #F5F5F5).

STT: first column, centered, automatically increases according to page.

Hover: line transition background #EFEBE9.

Card (react-bootstrap):

Light shadow (box-shadow: 0 2px 6px rgba(0,0,0,0.1))

Border thin 1px solid #E0E0E0

Card Header: large title, font-size: 18px, font-weight: 600.

Card Body: contains main content - detailed information

Background color: #F5F5F5

Show message:

When input is incorrect: Input border turns red (#E53935), shows a small error message below the input (font-size: 12px, italic, red).

When operation is successful: Show green Toast (#43A047) with content like "Added plan successfully" or "Updated information successfully".

Error Page"

Error 404: Displays the message "The page you are looking for does not exist". There is a button to return to the home page.

Error 500: Displays the message "A system error occurred. Please try again later."

Error pages are defined in the routes /error/404 and /error/500.

Pagination:

Use Pagination of react-bootstrap to perform

Placed at the end of a table, card list or page

Main color: B17d74, background color: white

Routing

Library: react-router-dom v6.

Route structure:

/login – login page

/home– overview page

/menu – menu dashboard

/finance – finance dashboard

/order– order dashboard

/ingredient – ingredient management dashboard

/report – report dashboard

Authentication: Based on JWT Token

State Management: use React Context API - sharing user information

Naming Convention

Component: PascalCase (e.g., UserList, PlanDetailModal).

CSS class: kebab-case (e.g., .table-header, .form-input-error).

React file: identical to the component name (UserList.jsx, LoginForm.jsx).

State and variables: camelCase (e.g., userData, isLoading)

1.1.2 Back-End

The backend system is built according to the Layered Architecture model with 4 main layers:

Controller Layer : Receives requests from the frontend, processes navigation logic and returns the corresponding response.

Service Layer: Contains business logic, processes data before communicating with the repository.

Repository Layer: Works directly with the database via JPA/Hibernate. Interface extends JpaRepository

Entity Layer: Describes the data structure corresponding to the table in the database. Mapping using annotations @Entity, @Table

RESTful API Standard

API is designed according to RESTful principles:

GET – Get data

POST – Create new

PUT/PATCH – Update

DELETE – Delete

Exception Handling

Create package exception:

- Use @RestControllerAdvice and @ExceptionHandler to handle global errors.

Security & Authentication

- Create package: security
- Authentication mechanism: JWT (JSON Web Token)
  - Token is generated after successful login.
  - Token contains username and roles information.
  - Save token in Header Authorization according to standard:
  - Authorization: Bearer <token>
- Authorization:
  - Permissions are defined through roles (ROLE_ADMIN,...).
  - Spring Security checks endpoint access rights @PreAuthorize("hasRole('ADMIN')")

Configuration Management:

- All configurations are stored in application.properties
- Use database MS SQL Driver

Coding Convention:

- Class names are capitalized and written together
- Variable and function names are camelCase

Logging & Monitoring:

- Using SLF4J

Transaction & Data Handling:

- Use @Transactional annotation at the Service layer to ensure data integrity.

DTO and Entity Mapping

- Using DTO (Data Transfer Object) to communicate between Controller and Front-end

Validation:

- Use validation dependency

Performance & Optimization

- Use Pagination when querying large lists (Spring Data Pageable).
- Use Lazy Loading for @OneToMany relationships.

1.2 Import Ingredient - Use Case

1.2.1.1 Class Diagram

Back-end class diagram for importing ingredients

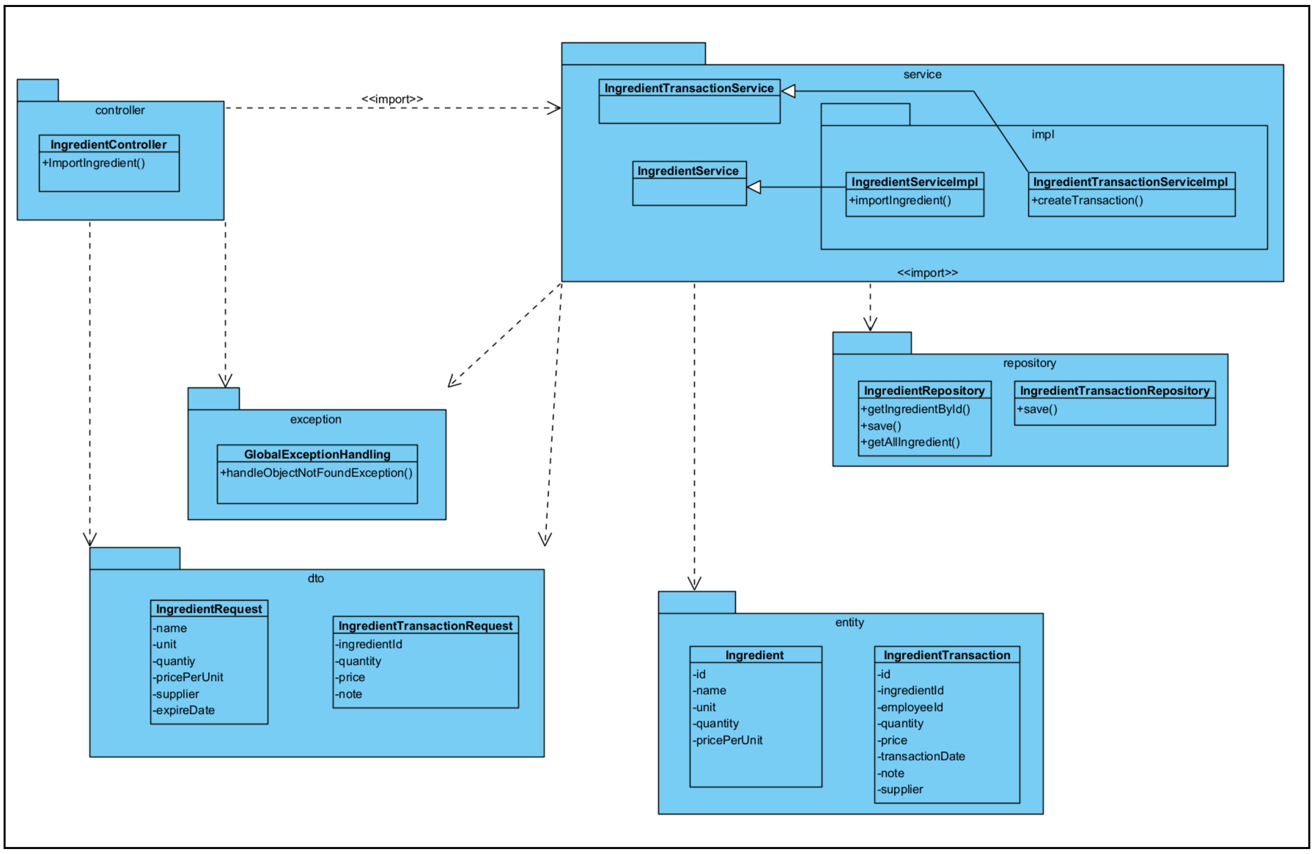


Figure 1: Diagram illustrates relationship between classes in each packages

The diagram above shows the classes in each package that participate in performing functions with the backend application. The files in the controller are used to receive and process requests with properties from the class in dto along with exceptions if any. The request will be called to the service to process and throw an exception if violated, storing database queries from the class and interface in the repository.

Front-end class diagram for importing ingredients

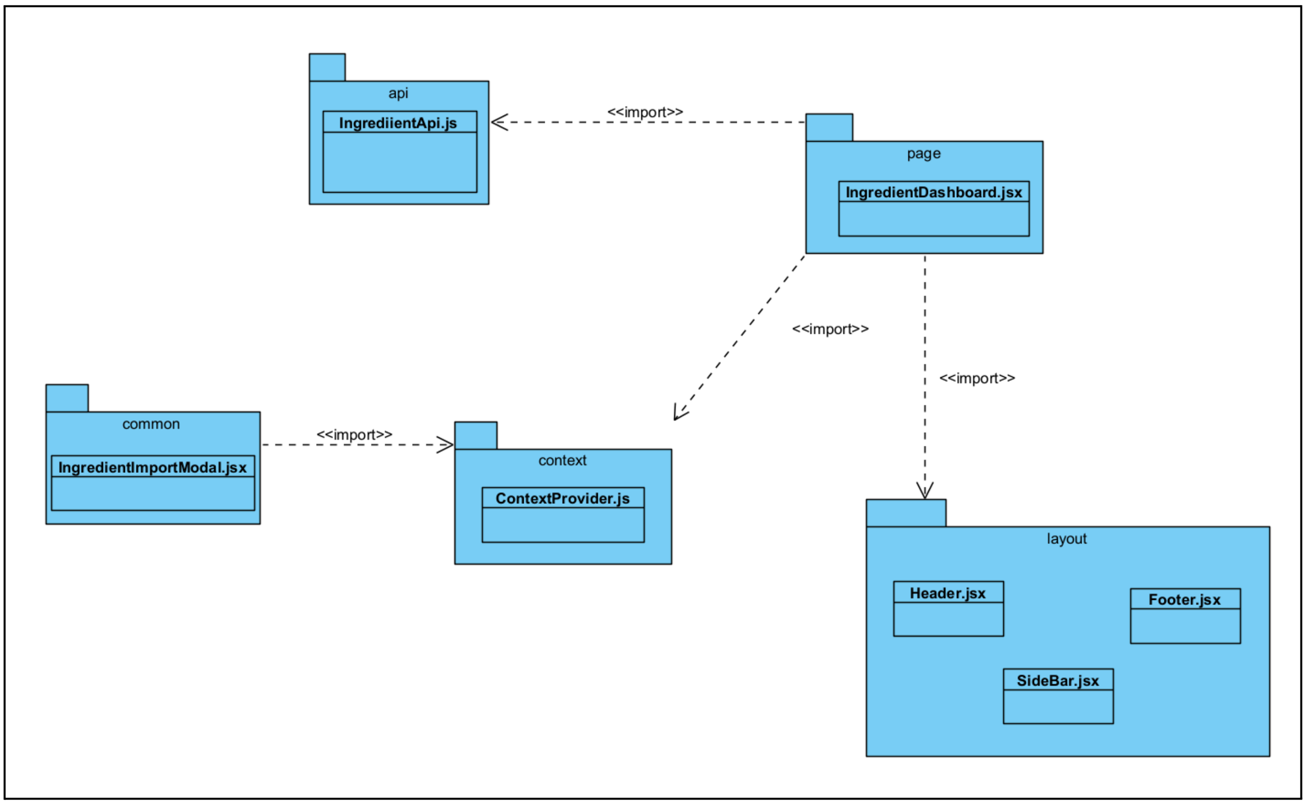


Figure 2: Diagram illustrates relationship between file jsx, file js in each packages

The diagram above shows the classes in each package that participate in performing functions with the frontend application. When the user wants to import ingredients, the user will start from the IngredientDashboard file in the page, then access the modal file to enter data information in common, then the data will be validated in the first layer on the frontend. Next, the data will be suggested by the method in the api file to connect to the backend to send and return data. The files in the context and layout perform additional tasks of retrieving data and forming the interface.

1.2.2 Class Description

1.2.2.1 Backend

Class	**IngredientController**
Description	This is the class to receive requests and return responses to users through the frontend and call the service to process information.
Base Class	None
Constructor	public IngredientController(IngredientController(IngredientService ingredientService, IngredientTransactionService ingredienttransactionService);

Prototype	@RestController @RequestMapping("/api/ingredients") public class IngredientController			
Source File	src/main/java/com.coffeemanagement/controller/IngredientController.java			
Namespace	/com.coffeemanagement/controller			
Attributes	**Name**	**Type**	**Description**	
	ingredientService	IngredientService	logic processing class for ingredient	
	ingredienttransactionService	IngredientTransactionService	logic processing class for ingredientTransaction	
Methods	**Name**	**Input**	**Output**	**Description**
	importIngredient	IngredientRequest	ResponseEntity <ApiResponse>	processing of importing ingredient requests

Class	**IngredientService**		
Description	interface of software for importing ingredient - update ingredient about quality		
Base Class	None		
Constructor	None		
Prototype	public interface IngredientService		
Source File	src/main/java/com.coffeemanagement/service/IngredientService.java		
Namespace	/com.coffeemanagement/service		
Attributes	**Name**	**Type**	**Description**
	None	None	None

Methods	Name	Input	Output	Description
	importIngredient	IngredientRequest	ApiResponse	It is a method to allow overriding of ingredient import processing.

Class	**IngredientServiceImpl**			
Description	business processing class of software for importing ingredient - update ingredient about quality			
Base Class	IngredientService			
Constructor	public IngredientServiceImpl(IngredientRepository ingredientRepository);			
Prototype	@Service public class IngredientServiceImpl implements IngredientService			
Source File	src/main/java/com.coffeemanagement/service/impl/IngredientServiceImpl.java			
Namespace	/com.coffeemanagement/service/impl			
Attributes	**Name**	**Type**	**Description**	
	ingredientRepository	IngredientRepository	interface class for accessing data from database	
Methods	**Name**	**Input**	**Output**	**Description**
	importIngredient	IngredientRequest	ApiResponse	Returns the status and message that updated the quantity state of ingredients

Class	**IngredientTransactionService**			
Description	interface of software for importing ingredient - create transaction			
Base Class	None			
Constructor	None			
Prototype	public interface IngredientTransactionService			
Source File	src/main/java/com.coffeemanagement/service/IngredientTransactionService.java			
Namespace	/com.coffeemanagement/service			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	importIngredient	IngredienttransactionRequest	ApiResponse	It is a method to allow overriding of ingredient import processing.

Class	**IngredientTransactionServiceImpl**		
Description	business processing class of software for importing ingredient - save transaction		
Base Class	IngredientTransactionService		
Constructor	public IngredientTransactionServiceImpl(IngredientTransactionRepository ingredienttransactionRepository);		
Prototype	@Service public class IngredientTransactionServiceImpl implements IngredientTransactionServic		
Source File	src/main/java/com.coffeemanagement/service/impl/IngredientTransactionServiceImpl.java		
Namespace	/com.coffeemanagement/service/impl		
Attributes	**Name**	**Type**	**Description**

	ingredientTransactionRepository	IngredientTransactionRepository	interface class for accessing data from database for transaction	
Methods	**Name**	**Input**	**Output**	**Description**
	createTransaction	IngredienttransactionRequest	ApiResponse	Returns the status and message that create transaction.

Class	**GlobalExceptionHandling**			
Description	class to catch all errors for business function handling			
Base Class	None			
Constructor	None			
Prototype	@RestControllerAdvice public class GlobalExceptionHandling			
Source File	src/main/java/com.coffeemanagement/exception/GlobalExceptionHandling.java			
Namespace	/com.coffeemanagement/exception			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	handleObjectNotFoundException	ObjectNotFoundException	ApiResponse	Catch return errors to prevent program crashes

Class	**IngredientTransactionRepository**
Description	interface of software for importing ingredient - query, retrieve data from database for transaction
Base Class	JpaRepository

Constructor	None			
Prototype	public interface IngredientTransactionRepository			
Source File	src/main/java/com.coffeemanagement/repository/IngredientTransactionRepository.java			
Namespace	/com.coffeemanagement/repository			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	createTransaction	IngredientTransaction	IngredientTransaction	It is a method to allow overriding of create transaction for process about updating

Class	**IngredientRepository**			
Description	interface of software for importing ingredient - query, retrieve data from database for ingredient			
Base Class	JpaRepository			
Constructor	None			
Prototype	public interface IngredientRepository			
Source File	src/main/java/com.coffeemanagement/repository/IngredientRepository.java			
Namespace	/com.coffeemanagement/repository			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**

	updateIngredient	Ingredient	Ingredient	It is a method to allow overriding of update quantity of ingredient from data

1.2.3 Screen Design

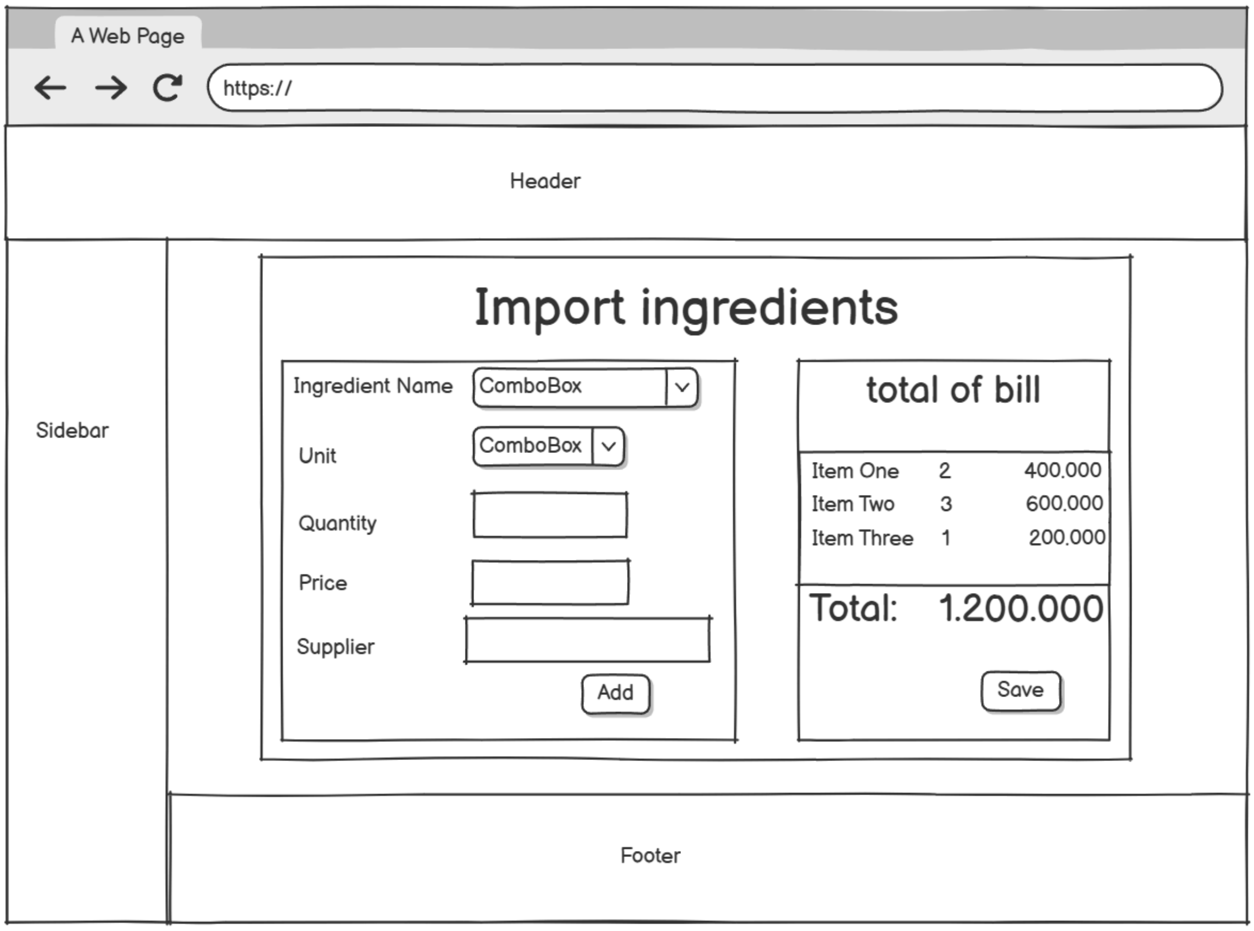


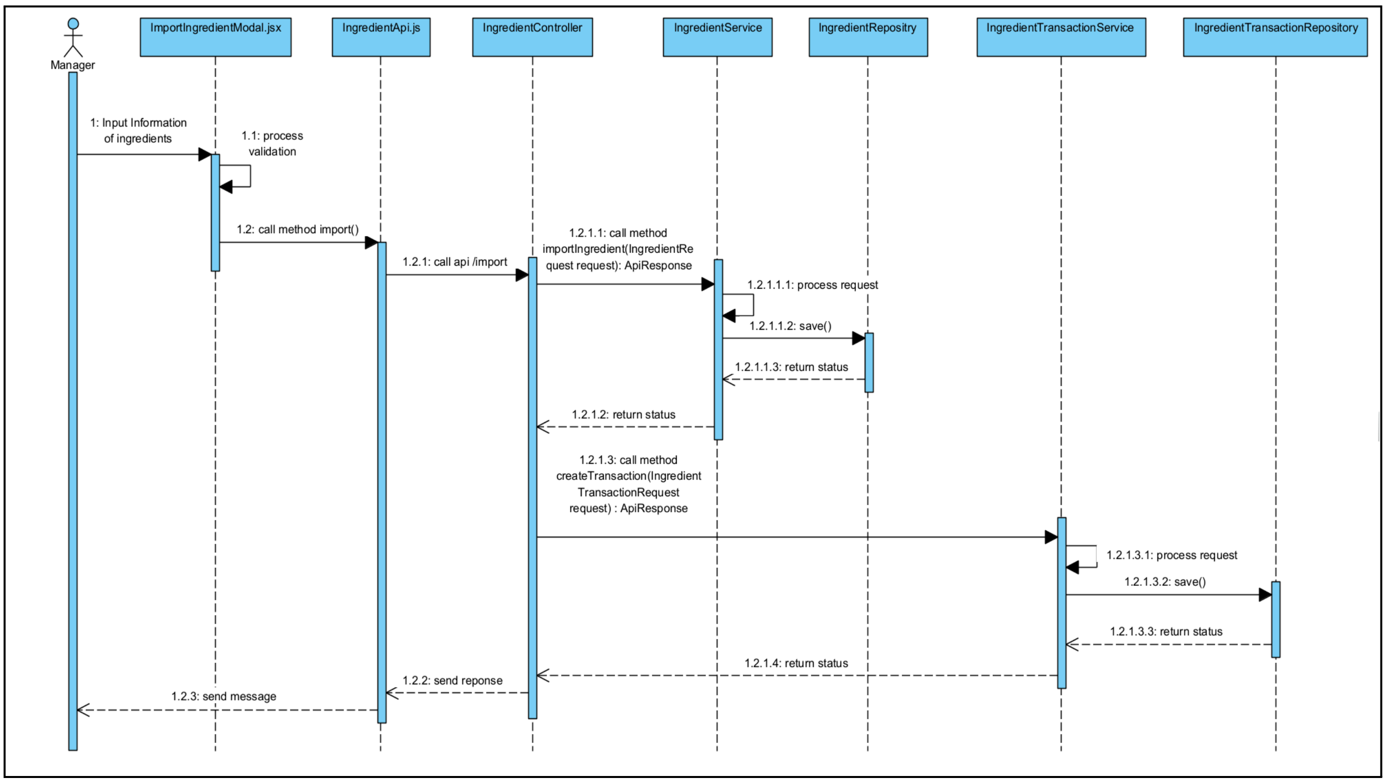
Figure 3: Screen for importing ingredient use case

For the importing ingredient screen, there will be 2 parts: 1 part to select the information of 1 ingredient and 1 side is the all of the ingredients for that invoice. After entering each ingredient, after clicking the 'add' button, the information will be saved to the total of the ingredients to process the transaction. After adding everything, the user can click the 'save' button to save and update the information to send to the backend.

No	Object/ control name	Type	Required	Length	Description
1	ingredientName	text	true	255	get the information for the ingredient name in the backend.

2	unit	text	true		get the information for the unit of measurement by bag or kg, ... in the backend.
3	quantity	text	true		Get information of ingredient amount then parseInt
4	price	text	true		Get information of ingredient price then convert to digital form
5	supplier	text	true		Get information of ingredient supplier - name, phone

1.2.4 Logic business process for importing ingredient



Manager
ImportIngredientModal.jsx
IngredientApi.js
IngredientController
IngredientService
IngredientRepositry
IngredientTransactionService
IngredientTransactionRepository
1: Input Information of ingredients
1.1: process validation
1.2: call method import()
1.2.1: call api /import
1.2.1.1: call method importIngredient(IngredientRequest request): ApiResponse
1.2.1.1.1: process request
1.2.1.1.2: save()
1.2.1.1.3: return status
1.2.1.2: return status
1.2.1.3: call method createTransaction(IngredientTransactionRequest request) : ApiResponse
1.2.1.3.1: process request
1.2.1.3.2: save()
1.2.1.3.3: return status
1.2.1.4: return status
1.2.2: send reponse
1.2.3: send message

Figure 4: Sequence diagram for importing ingredient use case with frontend and backend

In the diagram above, starting when the user enters information about ingredients into the ImportIngredientModal.js file, it will process validation and then call the import() function in IngredientApi.js to forward data to the backend by calling the API. The controller class will receive data and transfer data to the IngredientService class to process the business logic of importing ingredients with the importIngredient function and save data to the database through the IngredientRepository class. After successfully responding to the controller, it will call the IngredientTransactionService class to create and manage transactions and then call the IngredientTransactionRepository class to save the database.

1.3 Input Customer’s Order Use case

1.3.1 Class Diagram

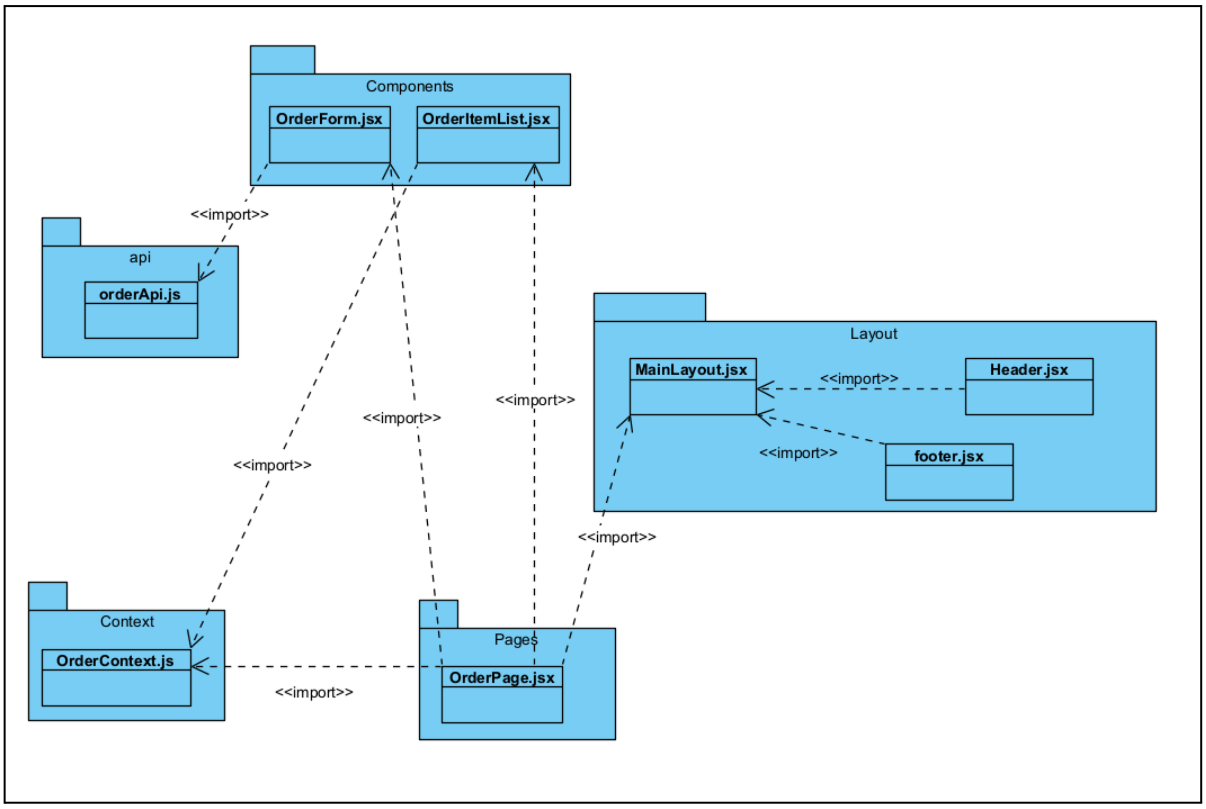


Figure 5: Diagram illustrates the relationship between the jsx, js in each package for the input customer’s order

This component diagram illustrates the front-end architecture for the order creation feature, organized by packages like pages, components, and api. It highlights the key dependencies, showing how the main OrderPage assembles various UI components and uses OrderContext for state management and orderApi for server communication.

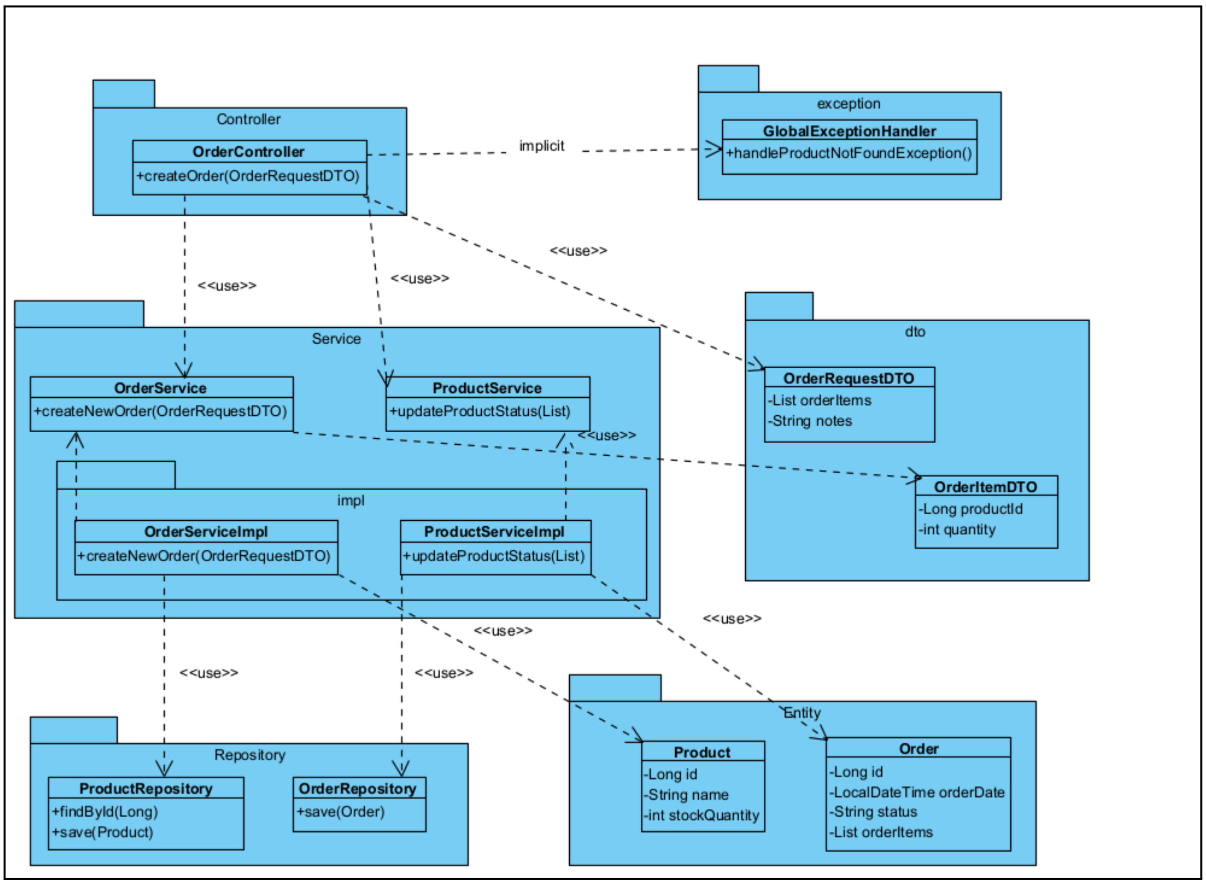


Figure 6: Diagram illustrates relationship between classes in each packages for Input Customer's Order

This diagram details the back-end's layered architecture for processing a new order, showcasing the clear separation of concerns across controller, service, repository, dto, and entity packages. It visualizes the dependency flow where the OrderController receives a DTO, uses Service interfaces for business logic, and how exceptions are managed by a GlobalExceptionHandler.

1.3.2 Class Description

1.3.2.1 FrontEnd

Class	**OrderItemList**
Description	A React component responsible for displaying the list of all items that have been added to the current order. It provides a running summary for the staff and allows for item removal or quantity adjustments.
Base Class	React.Component
Constructor	constructor(props)

Prototype	class OrderItemList extends React.Component			
Source File	src/components/OrderItemList.jsx			
Namespace	components			
Attributes	**Name**	**Type**	**Description**	
	orderItems	Array	The list of items in the current order. This data is retrieved from the shared OrderContext.	
	totalAmount	Number	The calculated total price of all items in the order, also retrieved from OrderContext.	
Methods	**Name**	**Input**	**Output**	**Description**
	handleRemoveItem	itemId	void	An event handler that is triggered when the user clicks the "remove" button for an item. It invokes the removeItemFromOrder function provided by the OrderContext.
	handleUpdateQuantity	itemId, newQuantity	void	An event handler for changing an item's quantity. It invokes the updateItemQuantity function from the OrderContext.

Class	**OrderContext (Provider/Consumer)**		
Description	The React Context provider for the order creation feature. It encapsulates the shared state (such as the list of items in the current order) and the functions to manipulate that state, making it available to any child component without prop drilling.		
Base Class	None (Created via React.createContext)		
Constructor	None		
Prototype	None		
Source File	src/context/OrderContext.js		
Namespace	context		
Attributes	**Name**	**Type**	**Description**

	orderItems	Array	The central state representing the list of all items currently in the order being created.	
	totalAmount	Number	The calculated total price, which is derived and updated whenever orderItems changes.	
	Name	**Input**	**Output**	**Description**
	addItemToOrder	product	void	Contains the logic to add a new product to the orderItems state. It handles both adding a new item and incrementing the quantity if the item already exists.
	removeItemFromOrder	itemId	void	Contains the logic to remove an item from the orderItems state using its unique ID.
	updateItemQuantity	itemId, newQuantity	void	Contains the logic to find an item by its ID in the state and update its quantity.
Methods	clearOrder	-	void	A function to reset the context's state (e.g., orderItems to an empty array) after an order is successfully submitted.

Class	**OrderForm**		
Description	A React component responsible for rendering the form where staff select products, adjust quantities, and submit a new order. It manages the local state of the order being created.		
Base Class	React.Component		
Constructor	constructor(props)		
Prototype	class OrderFormComponent extends React.Component		
Source File	src/components/OrderForm.jsx		
Namespace	components		
	Name	**Type**	**Description**
	menuItems	Array	List of available products for selection, passed as props.

Attributes

	currentOrder	Object	The order object being built, retrieved from context.	
	notes	String	Local state for storing special instructions for the order.	
	isLoading	boolean	Local state flag to indicate if the form is currently submitting data to the API.	
Methods	**Name**	**Input**	**Output**	**Description**
	handleSubmit	event	void	Gathers form data from state, calls orderApi.createOrder(), and handles the resulting promise (success/error).
	handleItemSelect	product	void	Adds a selected product to the currentOrder state via the context.
	handleQuantityChange	itemId, quantity	void	Updates the quantity of a specific item in the currentOrder state.
	handleNotesChange	event	void	Updates the notes state as the user types.

Class	**orderApi (Module)**			
Description	A JavaScript module that serves as the API Service layer. It encapsulates all HTTP request logic for communicating with the back-end's order-related endpoints, using a library like Axios or Fetch.			
Base Class	None (It's a module, not a class)			
Constructor	None			
Prototype	None			
Source File	src/api/orderApi.js			
Namespace	api			
Attributes	**Name**	**Type**	**Description**	
	apiClient	AxiosInstance	A pre-configured instance of Axios with the base URL and default headers (e.g., for authorization).	
Methods	**Name**	**Input**	**Output**	**Description**
	createOrder	orderData	Promise<Api	Sends a POST request containing

		(Object)	Response>	the orderData to the /api/orders endpoint. Returns a Promise that resolves with the server's response.
	getOrderById	orderId (String/Number)	Promise<ApiResponse>	Sends a GET request to /api/orders/{orderId} to fetch details of a specific order.
	updateOrderStatus	orderId, status	Promise<ApiResponse>	Sends a PUT or PATCH request to /api/orders/{orderId}/status to update the status of an order.

1.3.2.2 BackEnd

Class	**ProductService (Interface)**			
Description	Defines the contract for business logic operations related to products. This interface decouples the controller layer from the concrete implementation, adhering to SOLID principles and allowing for easier testing and maintenance.			
Base Class	None			
Constructor	N/A (It's an interface)			
Prototype	public interface ProductService			
Source File	src/main/java/com/csms/service/ProductService.java			
Namespace	com.csms.service			
Attributes	**Name**	**Type**	**Description**	
	N/A	N/A	Interfaces do not have attributes (fields). They may contain constants.	
Methods	**Name**	**Input**	**Output**	**Description**
	updateProductStatus	List<OrderItemDTO> items	void	Declares the method for updating the status or stock of products based on the items included in a newly created order.
	findById	Long productId	Product	Declares the method for retrieving a single product by its unique identifier.
	getAllProducts	-	List<Product>	Declares the method for fetching a list of all available products for the menu.

Class	**ProductServiceImpl**			
Description	Implements the business logic related to products, such as retrieving product information or updating stock levels. It is called by other services or controllers when order operations affect the product inventory.			
Base Class	Implements ProductService			
Constructor	public ProductServiceImpl(ProductRepository productRepository)			
Prototype	@Service public class ProductServiceImpl implements ProductService			
Source File	src/main/java/com/csms/service/impl/ProductServiceImpl.java			
Namespace	com.csms.service.impl			
Attributes	**Name**	**Type**	**Description**	
	productRepository	ProductRepository	A private, final field for accessing and persisting Product entities in the database.	
Methods	**Name**	**Input**	**Output**	**Description**
	updateProductStatus	List<OrderItemDTO> items	void	Iterates through the items of a new order, finds the corresponding product in the database, validates stock, and updates its stockQuantity. Throws ProductNotFoundException if a product does not exist.
	findById	Long productId	Product	Retrieves a single product by its unique ID, primarily for validation purposes.

Class	**ProductRepository**
Description	A Spring Data JPA interface that defines the data access layer for the Product entity. It provides standard CRUD methods and allows for the definition of custom database queries related to products.
Base Class	Extends org.springframework.data.jpa.repository.JpaRepository<Product, Long>

Constructor	N/A (It's an interface managed by Spring)			
Prototype	@Repository public interface ProductRepository extends JpaRepository<Product, Long>			
Source File	src/main/java/com/csms/repository/ProductRepository.java			
Namespace	com.csms.repository			
Attributes	**Name**	**Type**	**Description**	
	N/A	N/A	Interfaces do not have attributes.	
Methods	**Name**	**Input**	**Output**	**Description**
	save	Product entity	Product	(Inherited) Persists a new or updates an existing Product entity in the database.
	findById	Long productId	Optional<Product>	(Inherited) Retrieves a Product by its primary key.
	findAll	-	List<Product>	(Inherited) Retrieves all Product entities, typically used for displaying the menu.

Class	**OrderController**		
Description	A REST controller that handles all incoming HTTP requests for order-related operations. It acts as the entry point for the backend, validating input and delegating business logic to the service layer.		
Base Class	None		
Constructor	public OrderController(OrderService orderService, ProductService productService)		
Prototype	@RestController @RequestMapping("/api/orders") public class OrderController		
Source File	src/main/java/com/csms/controller/OrderController.java		
Namespace	com.csms.controller		
Attributes	**Name**	**Type**	**Description**
	orderService	OrderService	A private, final field injected via the constructor to handle order-related business logic.
	productService	ProductService	A private, final field injected via the constructor to handle product-related logic.

	Name	Input	Output	Description
	createOrder	@RequestBody @Valid OrderRequestDTO	ResponseEntity<ApiResponse>	POST /: Creates a new order based on the provided request body. Returns HTTP 201 (Created) on success.
	getAllOrders	Pageable pageable	ResponseEntity<Page<OrderResponseDTO>>	GET /: Retrieves a paginated list of all orders.
	getOrderById	@PathVariable Long id	ResponseEntity<OrderResponseDTO>	GET /{id}: Fetches the details of a single order by its ID.
Methods	updateOrderStatus	@PathVariable Long id, @RequestBody StatusUpdateDTO	ResponseEntity<ApiResponse>	PUT /{id}/status: Updates the status of an existing order (e.g., from 'PENDING' to 'COMPLETED').

Class	**OrderServiceImpl**		
Description	Implements the core business logic for creating and managing orders. It coordinates with various repositories to perform database operations and ensures data integrity.		
Base Class	Implements OrderService		
Constructor	public OrderServiceImpl(OrderRepository orderRepository, ProductRepository productRepository)		
Prototype	@Service public class OrderServiceImpl implements OrderService		
Source File	src/main/java/com/csms/service/impl/OrderServiceImpl.java		
Namespace	com.csms.service.impl		
	Name	**Type**	**Description**
	orderRepository	OrderRepository	A private, final field for persisting and retrieving Order entities.
Attributes	productRepository	ProductRepository	A private, final field for accessing Product data, used for validation or inventory updates.

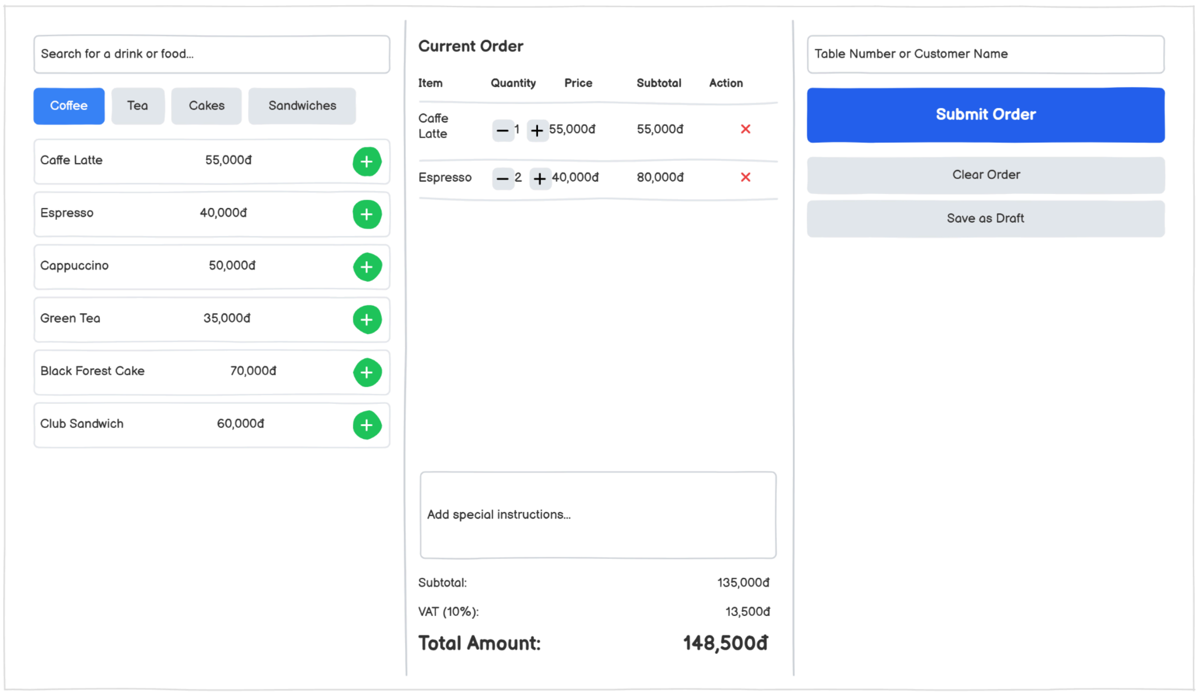
	Name	**Input**	**Output**	**Description**
	createNewOrder	OrderRequestDTO orderDTO	Order	Transforms the DTO into an Order entity, calculates the total price, sets the initial status to 'PENDING', validates product availability, and saves the order via the repository within a single transaction (@Transactional).
	findById	Long orderId	Optional<Order>	Retrieves an order by its unique identifier.
Methods	updateStatus	Long orderId, String newStatus	Order	Finds an existing order, validates the status transition, updates the status, and saves the changes.

Class	**OrderRepository**			
Description	A Spring Data JPA interface that defines the data access layer for the Order entity. It abstracts away the boilerplate code required for database operations, providing standard CRUD methods and the ability to define custom queries.			
Base Class	Extends org.springframework.data.jpa.repository.JpaRepository<Order, Long>			
Constructor	None (It's an interface managed by Spring)			
Prototype	@Repository public interface OrderRepository extends JpaRepository<Order, Long>			
Source File	src/main/java/com/csms/repository/OrderRepository.java			
Namespace	com.csms.repository			
	Name	**Type**	**Description**	
Attributes	None	None	Interfaces do not have attributes (fields).	
	Name	**Input**	**Output**	**Description**
	save	Order entity	Order	(Inherited) Persists a new or updates an existing Order entity in the database.
	findById	Long orderId	Optional<Order>	(Inherited) Retrieves an Order by its primary key.
	findAll	Pageable pageable	Page<Order>	(Inherited) Retrieves a paginated list of all Order entities.

Methods

	findByOrderDateBetween	LocalDateTime start, LocalDateTime end	List<Order>	(Custom) A custom query method to find all orders within a specific date range, used for reporting.

1.3.3 Screen Design


Search for a drink or food...
Coffee
Tea
Cakes
Sandwiches
Current Order
Table Number or Customer Name
Submit Order
Clear Order
Save as Draft
Add special instructions...

Caffe Latte	55,000đ
Espresso	40,000đ
Cappuccino	50,000đ
Green Tea	35,000đ
Black Forest Cake	70,000đ
Club Sandwich	60,000đ

Item	Quantity	Price	Subtotal	Action
Caffe Latte	1	55,000đ	55,000đ	×
Espresso	2	40,000đ	80,000đ	×

Subtotal:	135,000đ
VAT (10%):	13,500đ
Total Amount:	**148,500đ**

Figure 7: Screen Mockup for Input Customer's Order use case

No.	Object/Control Name	Type	Required	Length	Description
1	orderItems	Array<OrderItemDTO>	Yes	> 0	An array of objects representing the items in the order. Each object contains productId and quantity.
2	notes	String	No	255	Special instructions for the order, entered in the "Add special instructions..." text area.
3	tableNumber	String	No	50	The table number or customer name entered in the top-right input field.
4	employeeId	Long / UUID	Yes	-	The ID of the logged-in staff member creating the order. This is typically retrieved from the authentication state

					(e.g., JWT), not from a visible form field.

1.3.4 Logic business process for Input Customer’s Order

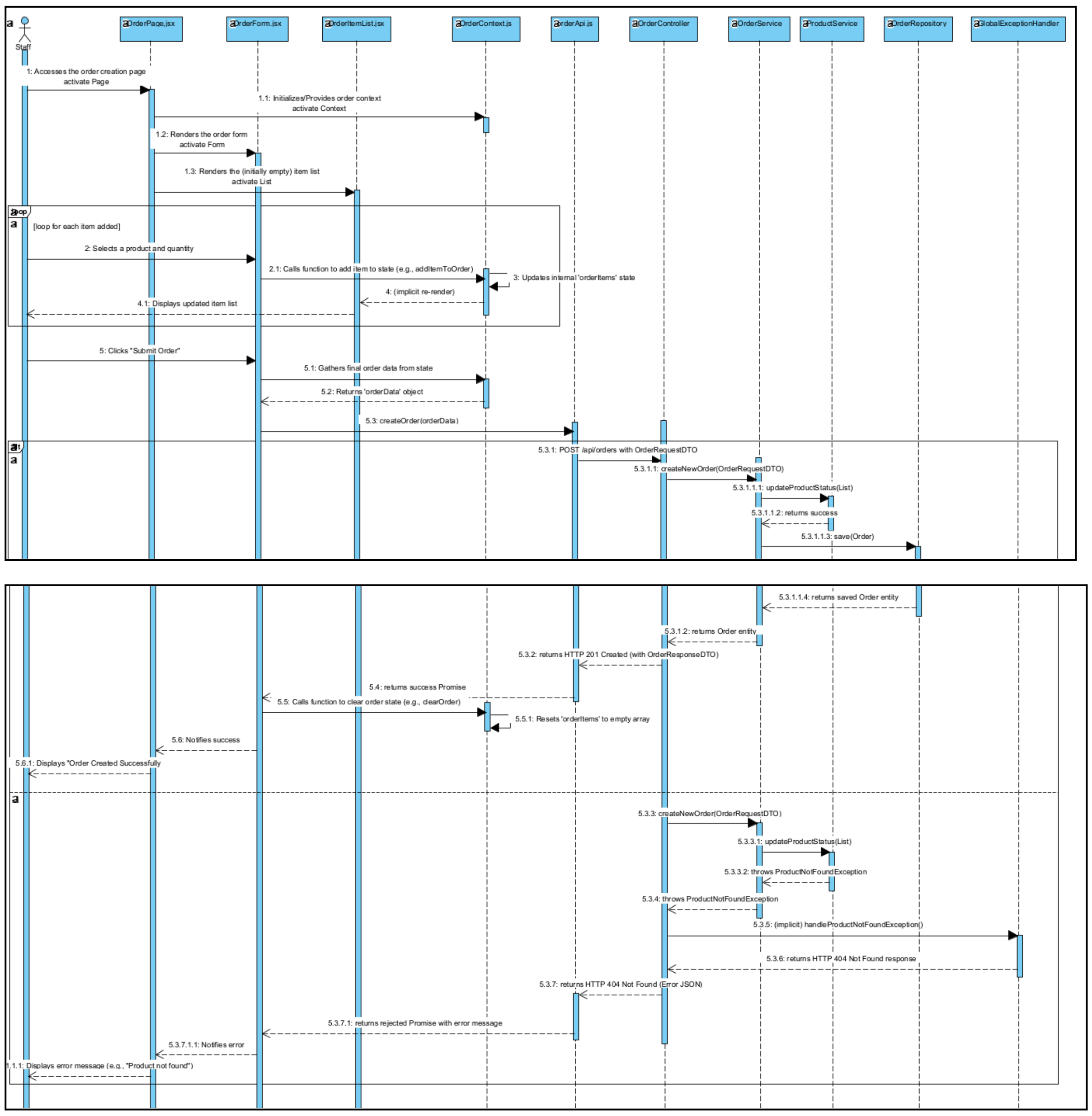

Staff
OrderPage.jsx
OrderForm.jsx
OrderItemList.jsx
OrderContext.js
orderApi.js
OrderController
OrderService
ProductService
OrderRepository
GlobalExceptionHandler
1: Accesses the order creation page activate Page
1.1: Initializes/Provides order context activate Context
1.2: Renders the order form activate Form
1.3: Renders the (initially empty) item list activate List
loop
[loop for each item added]
2: Selects a product and quantity
2.1: Calls function to add item to state (e.g., addItemToOrder)
3: Updates internal 'orderItems' state
4: (implicit re-render)
4.1: Displays updated item list
5: Clicks "Submit Order"
5.1: Gathers final order data from state
5.2: Returns 'orderData' object
5.3: createOrder(orderData)
alt
5.3.1: POST /api/orders with OrderRequestDTO
5.3.1.1: createNewOrder(OrderRequestDTO)
5.3.1.1.1: updateProductStatus(List)
5.3.1.1.2: returns success
5.3.1.1.3: save(Order)
5.3.1.1.4: returns saved Order entity
5.3.1.2: returns Order entity
5.3.2: returns HTTP 201 Created (with OrderResponseDTO)
5.4: returns success Promise
5.5: Calls function to clear order state (e.g., clearOrder)
5.5.1: Resets 'orderItems' to empty array
5.6: Notifies success
5.6.1: Displays "Order Created Successfully
5.3.3: createNewOrder(OrderRequestDTO)
5.3.3.1: updateProductStatus(List)
5.3.3.2: throws ProductNotFoundException
5.3.4: throws ProductNotFoundException
5.3.5: (implicit) handleProductNotFoundException()
5.3.6: returns HTTP 404 Not Found response
5.3.7: returns HTTP 404 Not Found (Error JSON)
5.3.7.1: returns rejected Promise with error message
5.3.7.1.1: Notifies error
.1.1: Displays error message (e.g., "Product not found")

Figure 8: Sequence diagram for Input Customer’s Order use case with frontend and backend

This sequence diagram provides an end-to-end realization of the "Input Customer's Order" use case, detailing interactions from the staff's UI actions to the back-end's RESTful processing. It illustrates the complete flow, including the front-end's state management loop for adding items, the stateless API call, and both the successful creation path and the error handling path for invalid products.

1.4 View menu- Use Case

1.4.1 Class Diagram

Back-end class diagram for view menu

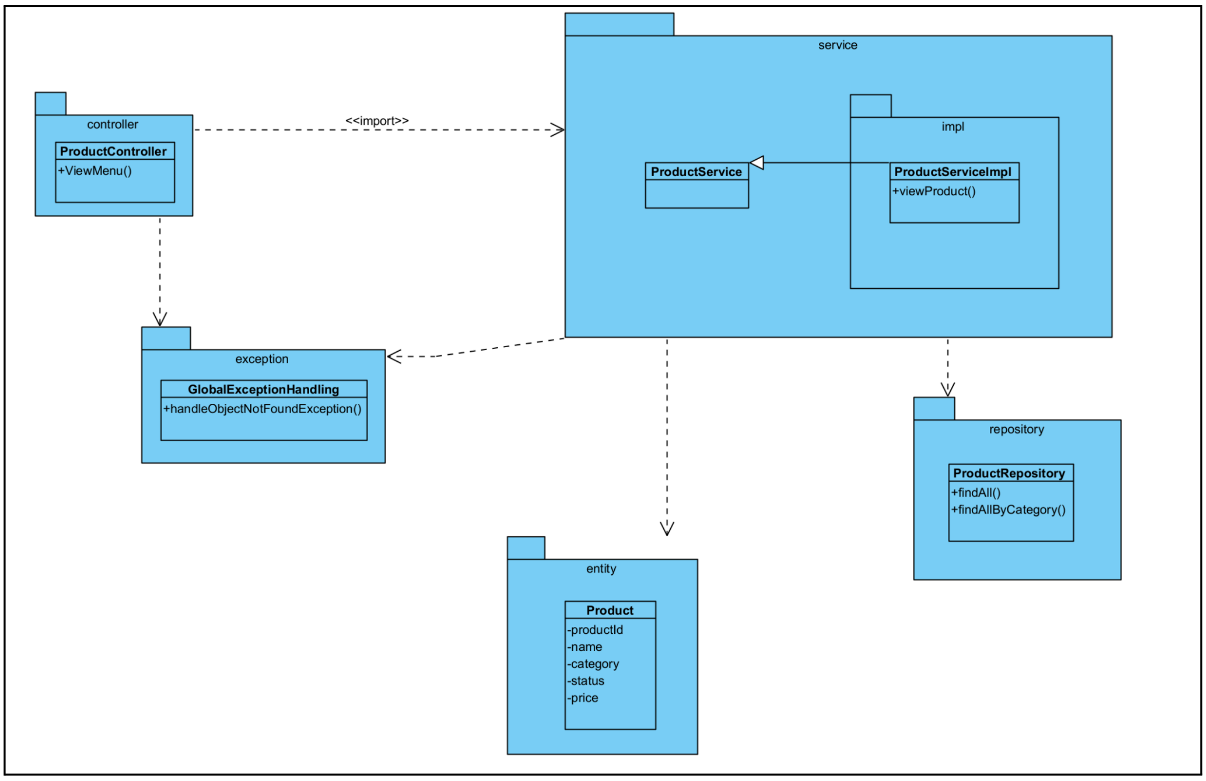


Figure 1: Diagram illustrates relationship between classes in each packages

The diagram above shows the classes in each package that participate in performing the View Menu function. The ProductController manages exceptions through GlobalExceptionHandling. The request is processed by ProductService which calls its implementations to handle business logic. Database queries are managed by classes and interfaces in the repository package, while entity classes define database structures for products.

Front-end class diagram for importing ingredients

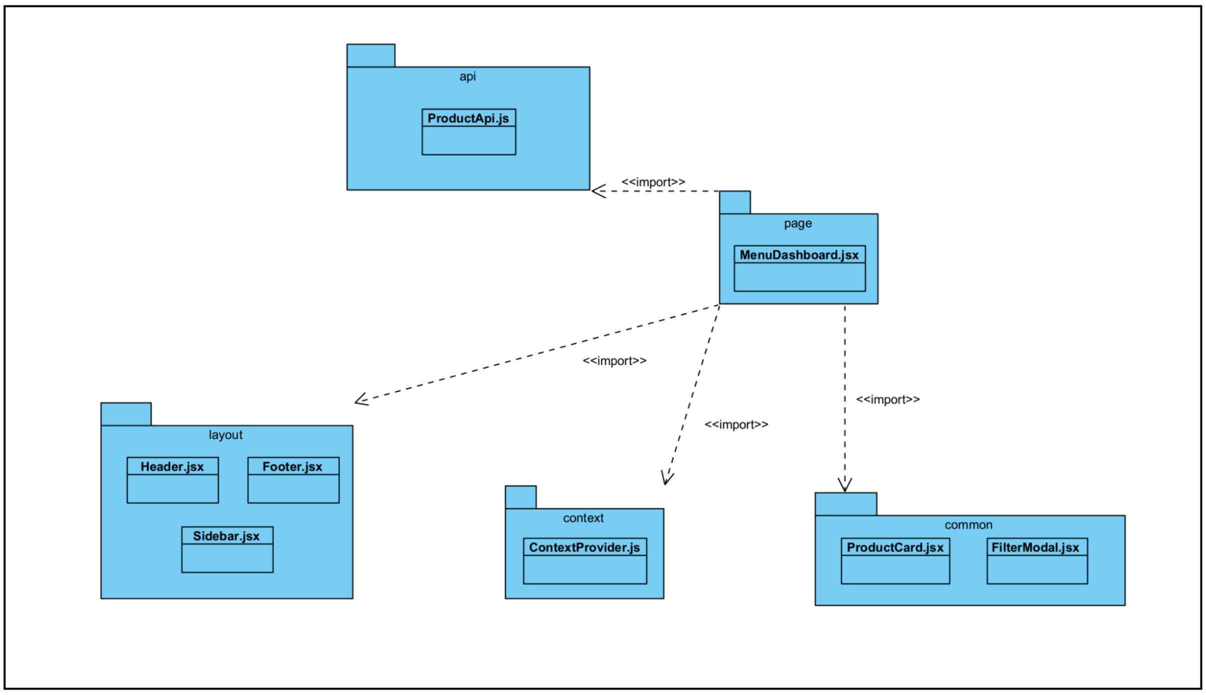


Figure 2: Diagram illustrates relationship between file jsx, file js in each packages

The diagram above shows the components in each folder that work together in the frontend application. When the user accesses the menu management feature, the process begins at the MenuDashboard.jsx file in the page folder. This file imports data handling functions from ProductApi.js in the api folder to interact with the backend. It also uses ProductCard.jsx, FilterModal.jsx from the common folder to display product information, filter products and ContextProvider.js from the context folder to manage shared state. Additionally, layout components such as Header.jsx, Footer.jsx, and Sidebar.jsx in the layout folder help build the main user interface and navigation structure.

1.4.2 Class Description

1.4.2.1 Backend

Class	**ProductController**
Description	This is the class to receive requests and return responses to users through the frontend and call the service to process information.
Base Class	None
Constructor	public ProductController(ProductController(ProductService productService);
Prototype	@RestController @RequestMapping("/api/products")

	public class ProductController			
Source File	src/main/java/com.coffeemanagement/controller/ProductController.java			
Namespace	/com.coffeemanagement/controller			
Attributes	**Name**	**Type**	**Description**	
	productService	ProductService	logic processing class for products	
Methods	**Name**	**Input**	**Output**	**Description**
	viewMenu	String	ResponseEntity <ApiResponse>	processing of get product data requests

Class	**ProductService**			
Description	interface of software for get product data			
Base Class	None			
Constructor	None			
Prototype	public interface ProductService			
Source File	src/main/java/com.coffeemanagement/service/Product/ProductService.java			
Namespace	/com.coffeemanagement/service			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	findAllByCategory	String	ApiResponse	It is a method to allow overriding of get product by category

Class	**ProductServiceImpl**			
Description	business processing class of software for get product data			
Base Class	ProductService			
Constructor	public ProductServiceImpl(ProductRepository productRepository);			
Prototype	@Service public class ProductServiceImpl implements ProductService			
Source File	src/main/java/com.coffeemanagement/service/impl/ProductServiceImpl.java			
Namespace	/com.coffeemanagement/service/impl			
Attributes	**Name**	**Type**	**Description**	
	productRepository	ProductRepository	interface class for accessing data from database for transaction	
Methods	**Name**	**Input**	**Output**	**Description**
	getProductByCategory	Category	ApiResponse	Returns the status and message that get product data

Class	**ProductRepository**			
Description	interface of software for retrieve data from database for product			
Base Class	JpaRepository			
Constructor	None			
Prototype	public interface ProductRepository			
Source File	src/main/java/com.coffeemanagement/repository/ProductRepository.java			
Namespace	/com.coffeemanagement/repository			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**

	findAll	None	Product	It is a method to allow overriding of find all products
	findAllByCategory	String	Product	It is a method to allow overriding of find all products by category

Class	**GlobalExceptionHandling**			
Description	class to catch all errors for business function handling			
Base Class	None			
Constructor	None			
Prototype	@RestControllerAdvice public class GlobalExceptionHandling			
Source File	src/main/java/com.coffeemanagement/exception/GlobalExceptionHandling.java			
Namespace	/com.coffeemanagement/exception			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	handleObjectNotFoundException	ObjectNotFound Exception	ApiResponse	Catch return errors to prevent program crashes

1.4.3 Screen Design

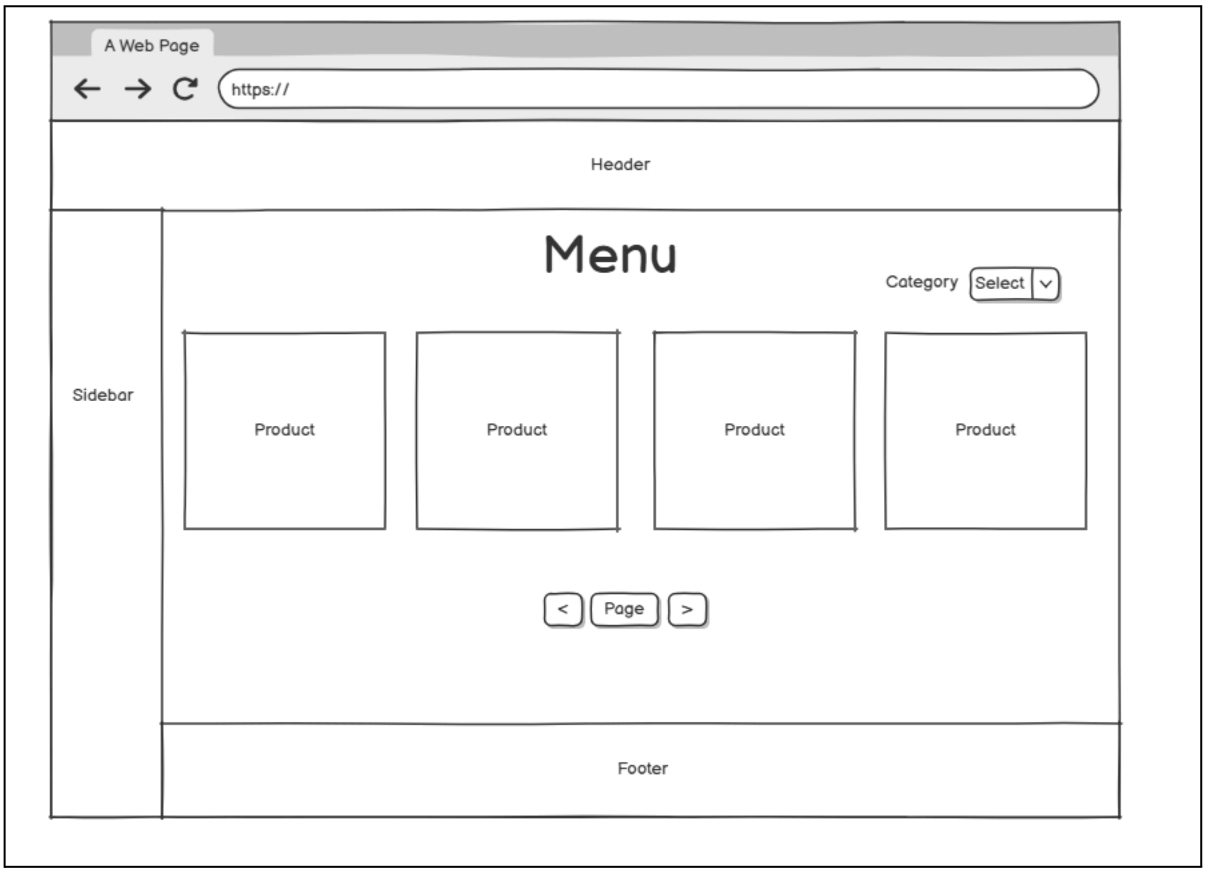


Figure 3: Screen for viewing menu use case

The view menu screen is designed with a clear and intuitive layout. At the top, the header displays the title "Menu" along with a category dropdown that allows users to filter products by type. On the left side, a sidebar provides navigation links to other main sections such as Home, Orders, and Reports. The main area showcases a grid of product cards, each displaying a product's image, name, and price. Below the grid, pagination controls enable users to navigate between different pages of products. At the bottom of the page, a footer contains general information or useful links. This layout helps users easily browse and select products while maintaining a consistent structure throughout the application.

No	Object/ control name	Type	Required	Length	Description
1	category	Dropdown (String)	true	255	**Category** selection dropdown: The User can select a specific category to filter products. If left blank, the system will default to calculating for all categories.
2	product	Object	true		get the information for the data of the product.

1.4.4 Logic business process for importing ingredient

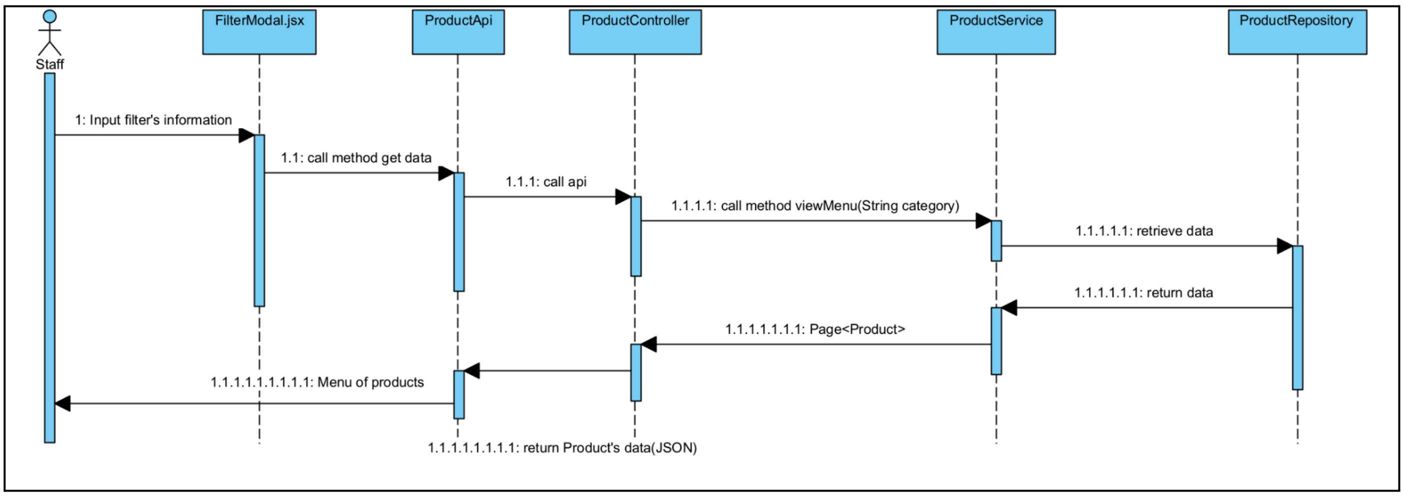

Staff
FilterModal.jsx
ProductApi
ProductController
ProductService
ProductRepository
1: Input filter's information
1.1: call method get data
1.1.1: call api
1.1.1.1: call method viewMenu(String category)
1.1.1.1.1: retrieve data
1.1.1.1.1.1: return data
1.1.1.1.1.1.1: Page<Product>
1.1.1.1.1.1.1.1.1: Menu of products
1.1.1.1.1.1.1.1: return Product's data(JSON)

Figure 4: Sequence diagram for view menu use case with frontend and backend

In the diagram above, the process begins when the staff inputs the filter information into the FilterModal.jsx file. The component then calls the method getData() from the ProductApi.js file to retrieve filtered products. The API file sends a request to the backend by calling the corresponding endpoint in the ProductController class. The controller then calls the viewMenu(String category) method in the ProductService class to handle the business logic and request data from the ProductRepository. After retrieving the data, it returns a Page<Product> object to the service, which then sends the result back to the controller. The controller converts this data into JSON format and sends it as a response to the frontend, where the product list (menu) is displayed to the user.

1.5 View Finance Dashboard - Use Case

1.5.1 Class diagram

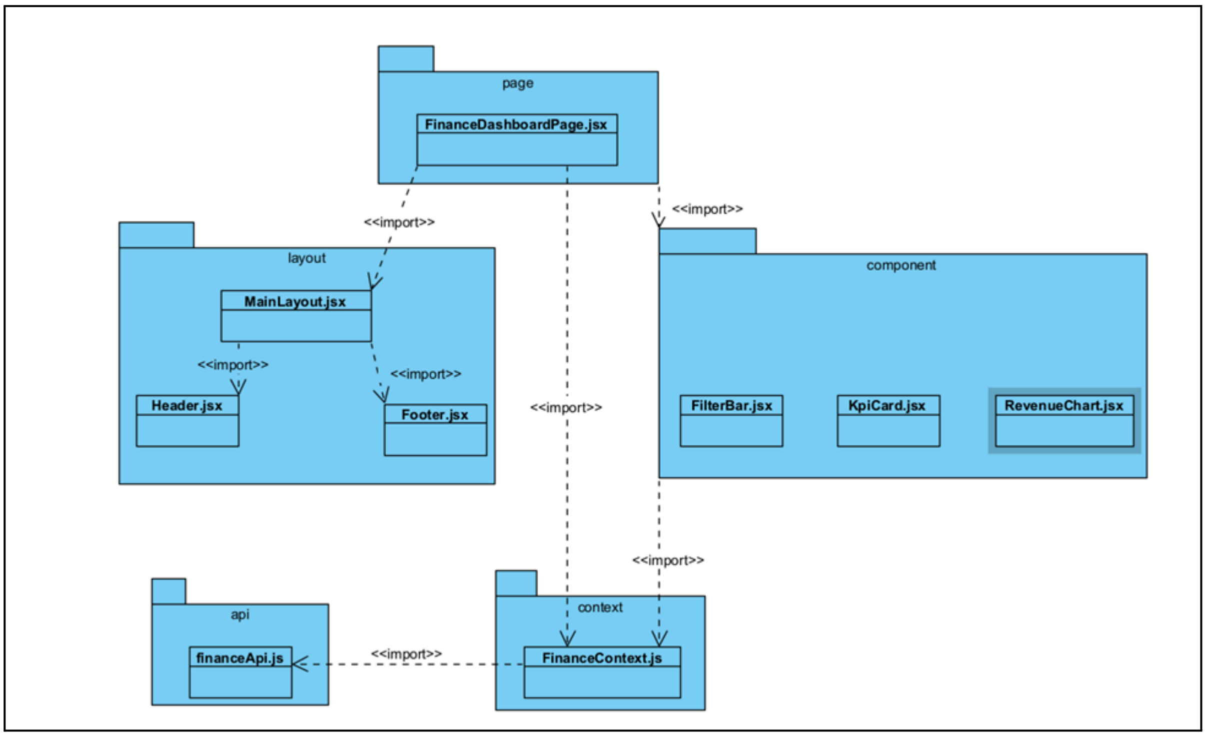


Figure 5: Diagram illustrates the relationship between the jsx, js in each package for View FinanceDashboard

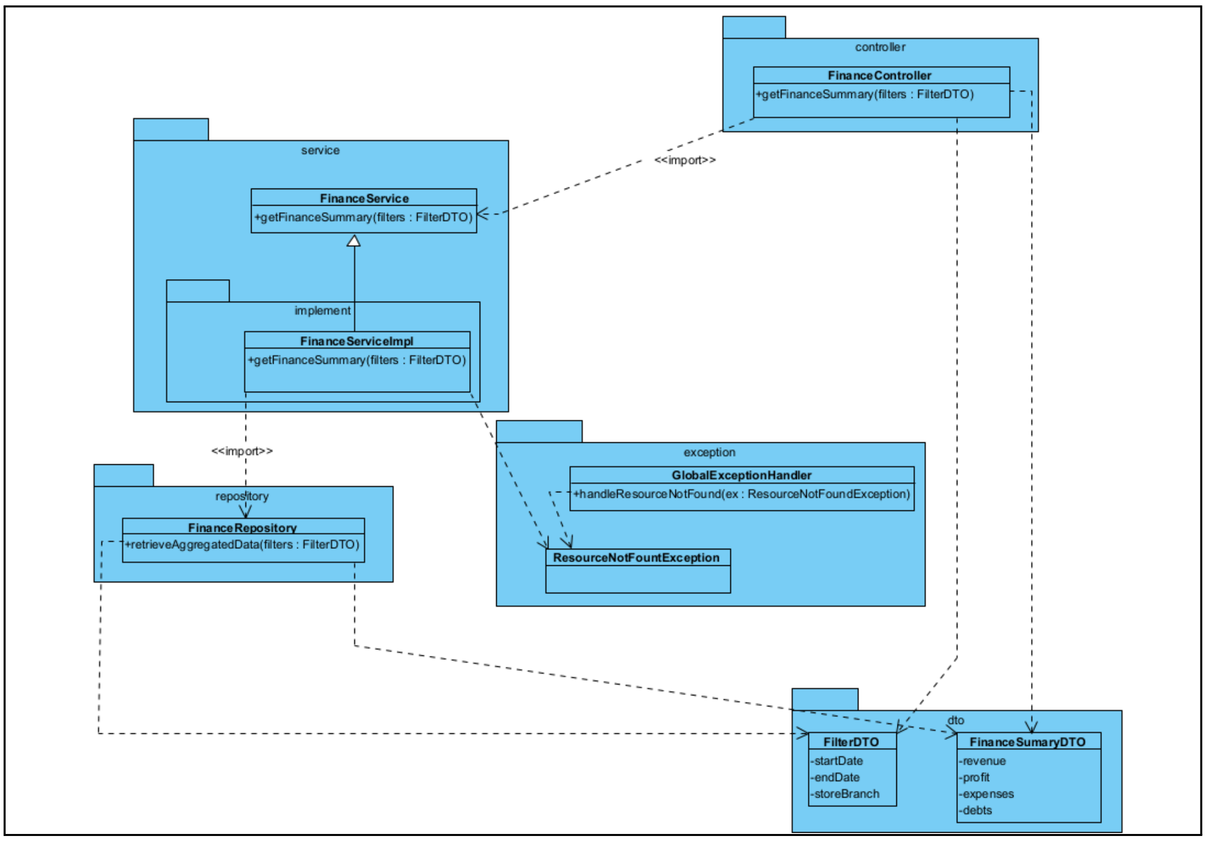

controller
FinanceController
+getFinanceSummary(filters : FilterDTO)
<<import>>
service
FinanceService
+getFinanceSummary(filters : FilterDTO)
implement
FinanceServiceImpl
+getFinanceSummary(filters : FilterDTO)
<<import>>
repository
FinanceRepository
+retrieveAggregatedData(filters : FilterDTO)
exception
GlobalExceptionHandler
+handleResourceNotFound(ex : ResourceNotFoundException)
ResourceNotFountException
dto
FilterDTO
-startDate
-endDate
-storeBranch
FinanceSumaryDTO
-revenue
-profit
-expenses
-debts

Figure 6: Diagram illustrates the relationship between classes in each package for View FinanceDashboard

1.5.2 Class Description

Class	**FinanceController**
Description	This class acts as an API Endpoint, receiving HTTP requests from the client
Base Class	None
Constructor	FinanceController(FinanceService financeService, SecurityService securityService)
Prototype	@RestController @RequestMapping("/api/finance") public class FinanceController
Source File	src/main/java/com.coffeemanagement/controller/FinanceController.java

Namespace	/com.coffeemanagement/controller			
Attributes	**Name**	**Type**	**Description**	
	financeService	FinanceService	Dependency for accessing financial business logic	
Methods	**Name**	**Input**	**Output**	**Description**
	getFinanceSummary	filters: @RequestBody FilterDTO	ResponseEntity< FinanceSummary DTO>	Allows the client to fetch aggregated financial data after validating access rights

Class	**FinanceService (Interface)**			
Description	An interface of software for getting finance data			
Base Class	None			
Constructor	N/A			
Prototype	public FinanceSummaryDTO getFinanceSummary(FilterDTO filters)			
Source File	src/main/java/com.coffeemanagement/service/FinanceService.java			
Namespace	/com.coffeemanagement/service			
Attributes	**Name**	**Type**	**Description**	
	N/A	**N/A**	**N/A**	
Methods	**Name**	**Input**	**Output**	**Description**
	getFinanceSummary	filters: FilterDTO	FinanceSummaryDTO	

Class	**FinanceServiceImpl**
Description	The implementation class for FinanceService, containing core business logic.

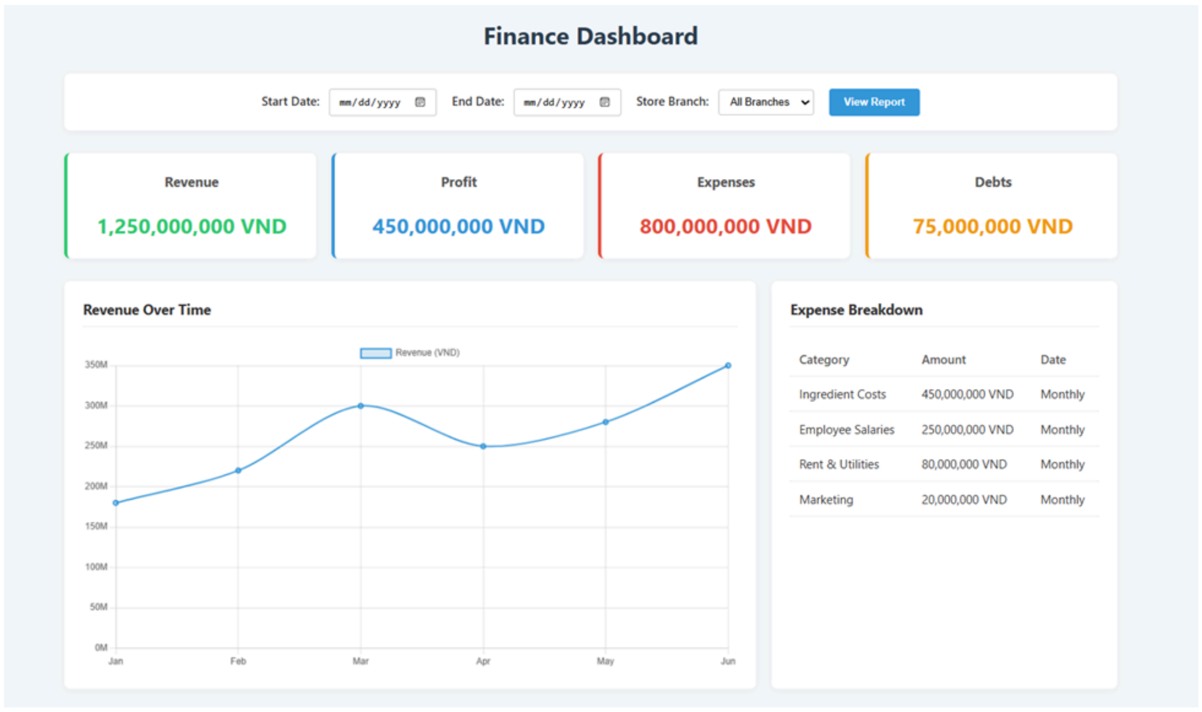
Base Class	None			
Constructor	Default constructor, All-arguments constructor			
Prototype	@Service public class FinanceServiceImpl implements FinanceService			
Source File	src/main/java/com.coffeemanagement/service/FinanceServiceImpl.java			
Namespace	/com.coffeemanagement/service			
Attributes	**Name**	**Type**	**Description**	
	financeRepository	FinanceRepository	Dependency for accessing the database.	
Methods	**Name**	**Input**	**Output**	**Description**
	getFinanceSummary	filters: FilterDTO	FinanceSummary DTO	Coordinates business logic, calls the Repository to retrieve and return financial data.

Class	**FinanceRepository (Interface)**			
Description	The Data Access Layer interface defines methods for querying financial data.			
Base Class	JpaRepository<T, ID>			
Constructor	N/A			
Prototype	@Repository public FinanceSummaryDTO retrieveAggregatedData(FilterDTO filters)			
Source File	src/main/java/com.coffeemanagement/repository/FinanceRepository.java			
Namespace	/com.coffeemanagement/repository			
Attributes	**Name**	**Type**	**Description**	
	N/A	**N/A**	**N/A**	
Methods	**Name**	**Input**	**Output**	**Description**

	retrieveAggregatedData	filters: FilterDTO	Optional<FinanceSummaryDTO>	Executes a complex query to aggregate data from the database and maps it to a DTO.

Class	**GlobalExceptionHandling**			
Description	class to catch all errors for business function handling			
Base Class	None			
Constructor	None			
Prototype	@RestControllerAdvice public class GlobalExceptionHandling			
Source File	src/main/java/com.coffeemanagement/exception/GlobalExceptionHandling.java			
Namespace	/com.coffeemanagement/exception			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	handleObjectNotFoundException	ObjectNotFoundException	ApiResponse	Catch return errors to prevent program crashes

1.5.3 Screen Design



Finance Dashboard

Start Date: mm/dd/yyyy End Date: mm/dd/yyyy Store Branch: All Branches View Report

Revenue	Profit	Expenses	Debts
1,250,000,000 VND	450,000,000 VND	800,000,000 VND	75,000,000 VND

Revenue Over Time

Expense Breakdown

Category	Amount	Date
Ingredient Costs	450,000,000 VND	Monthly
Employee Salaries	250,000,000 VND	Monthly
Rent & Utilities	80,000,000 VND	Monthly
Marketing	20,000,000 VND	Monthly

No	Object/ control name	Type	Required	Length	Description
1	startDate	Date Picker (String)	true	10	**Start Date** Selection Field: User selects the start date of the reporting period. The data is sent in "YYYY-MM-DD" format.
2	endDate	Date Picker (String)	true	10	**End Date** Selection Field: User selects the end date of the reporting period. The data is sent in "YYYY-MM-DD" format.
3	storeBranchId	Dropd own (Intege r)	true	N/A	**Branch** selection dropdown: The User can select a specific branch to view the report. If left blank, the system will default to calculating for all branches.
4	btnViewReport	Button	true	N/A	**"View Report" Button**: When the user clicks, values from the above controls will be collected and a request will be sent to Backend.

1.6 View Daily Report

1.6.1 Class diagram

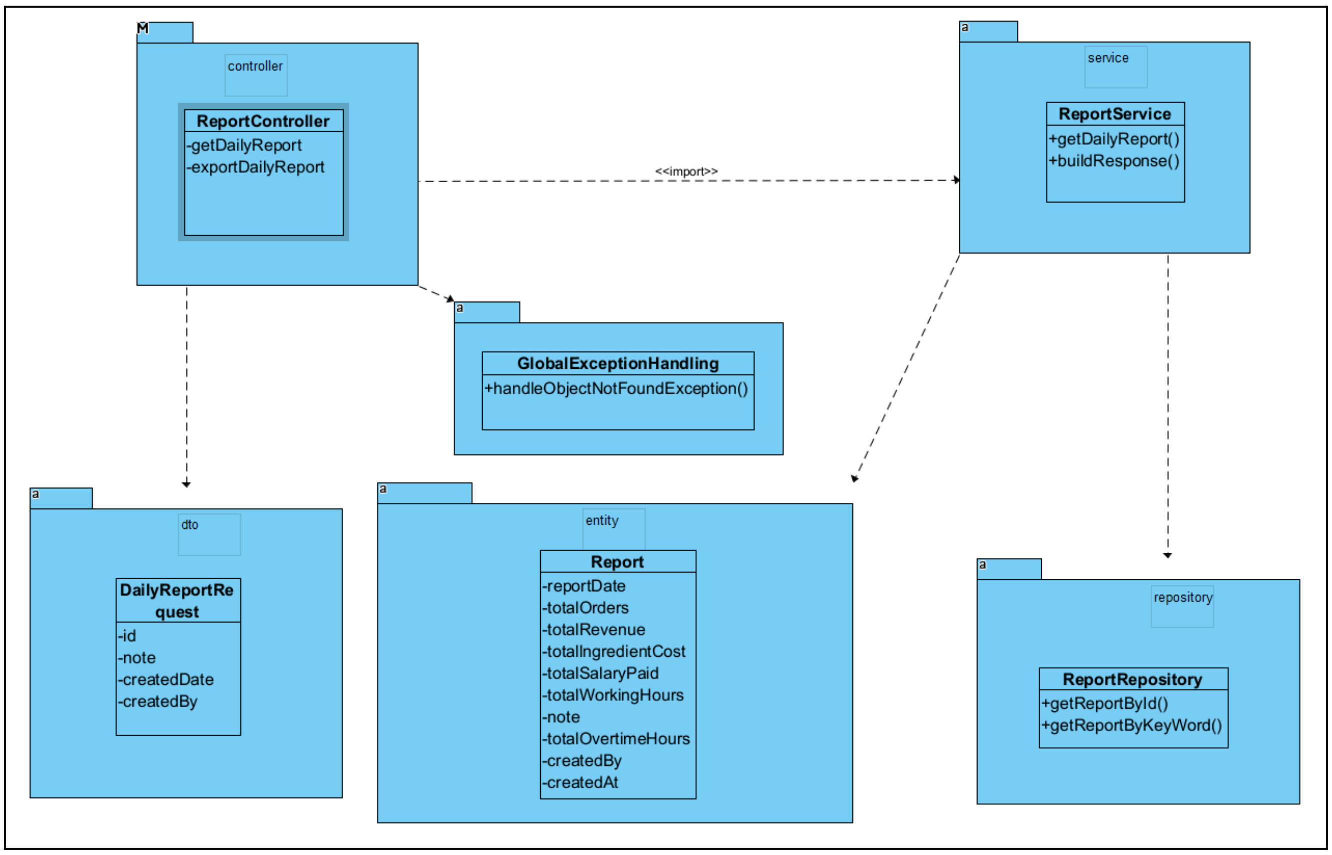


Figure 1: Diagram illustrates relationship between classes in each packages

The diagram above shows the classes in each package that participate in performing functions with the backend application. The files in the controller are responsible for receiving and processing requests using data transferred from the classes in dto, and handling exceptions if any occur. These requests are then passed to the service layer, where the business logic is processed, and exceptions are thrown when validation rules are violated. The repository package contains the classes and interfaces that handle database queries and interact directly with the data stored in the system.

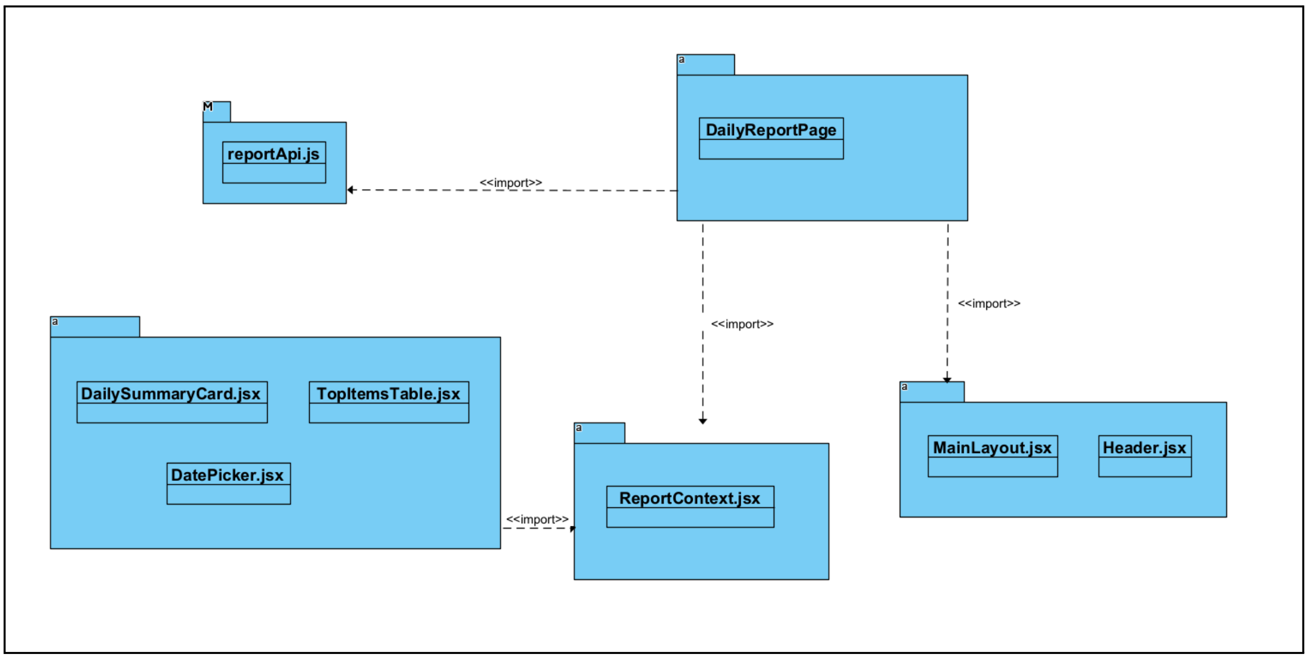


Figure 2: Diagram illustrates relationship between file jsx, file js in each packages

The diagram above shows the files in each package that participate in performing functions with the frontend application. When the user wants to view or export a daily report, the process begins from the DailyReportPage file in the pages package. This file interacts with components in the common package, such as tables and cards, to display data. The data is first validated on the frontend before being sent to the backend through methods defined in the api package. The context package manages the application state, storing and sharing report data between components, while the layout package organizes the user interface structure, including the header, sidebar, and main content area.

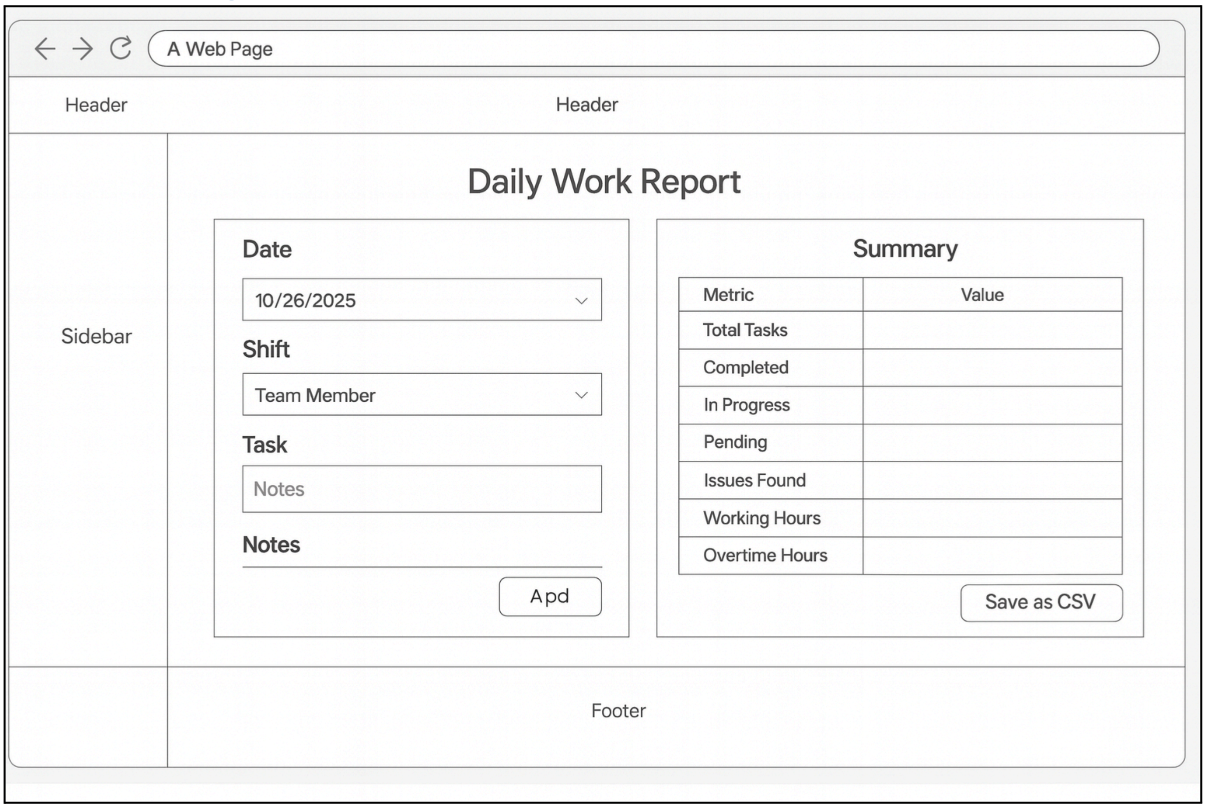
1.6.2 Class description

Class	**ReportController**			
Description	This class is used to receive requests and return responses to the frontend. It calls the service layer to handle logic for viewing and exporting daily reports.			
Base Class	None			
Constructor	public ReportController(ReportService reportService)			
Prototype	@RestController @RequestMapping("/api/reports") public class ReportController			
Source File	src/main/java/com/coffeemanagement/controller/ReportController.java			
Namespace	/com/coffeemanagement/controller			
Attributes	**Name**	**Type**	**Description**	
	none	none	none	
Methods	**Name**	**Input**	**Output**	**Description**
	exportDailyReport	String date	ResponseEntity<Resource>	Exports the daily report file (Excel/PDF).
	getDailyReport	String date	DailyReport	Returns daily report data for a given date.

Class	**ReportService**			
Description	This interface defines business logic methods for generating and retrieving daily report data.			
Base Class	None			
Constructor	none			
Prototype	public interface ReportService			
Source File	src/main/java/com.coffeemanagement/service/ReportService.java			
Namespace	/com.coffeemanagement/service			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	getDailyReport	String date	Report	Retrieves the daily report data for a given date.
	buildResponse	Report report	DailyReportResponse	Builds a response object for frontend display.

Class	ReportRepository			
Description	Repository interface for accessing report and sales-related data to generate daily reports.			
Base Class	JpaRepository			
Constructor	none			
Prototype	@Repository public interface ReportRepository extends JpaRepository<DailyReport, Long>			
Source File	src/main/java/com.coffeemanagement/repository/ReportRepository.java			
Namespace	/com/coffeemanagement/repository			
Attributes	**Name**	**Type**	**Description**	
	None	None	None	
Methods	**Name**	**Input**	**Output**	**Description**
	getReportById	String date	DailyReport	Find report by a specific date
	getReportByKeyWord	String keyword	List<Report>	Retrieves reports that match a search keyword.

1.6.3 Screen design



Metric	Value
Total Tasks	
Completed	
In Progress	
Pending	
Issues Found	
Working Hours	
Overtime Hours	

Figure 4: Screen for viewing daily report use case

For the Daily Work Report screen, there are two main sections:

The left side allows the manager to select a specific date, choose a shift or team member, and input the daily tasks and related notes.

The right side displays a summary table showing key daily metrics such as total tasks, completed tasks, tasks in progress, pending work, issues found, working hours, and overtime hours.

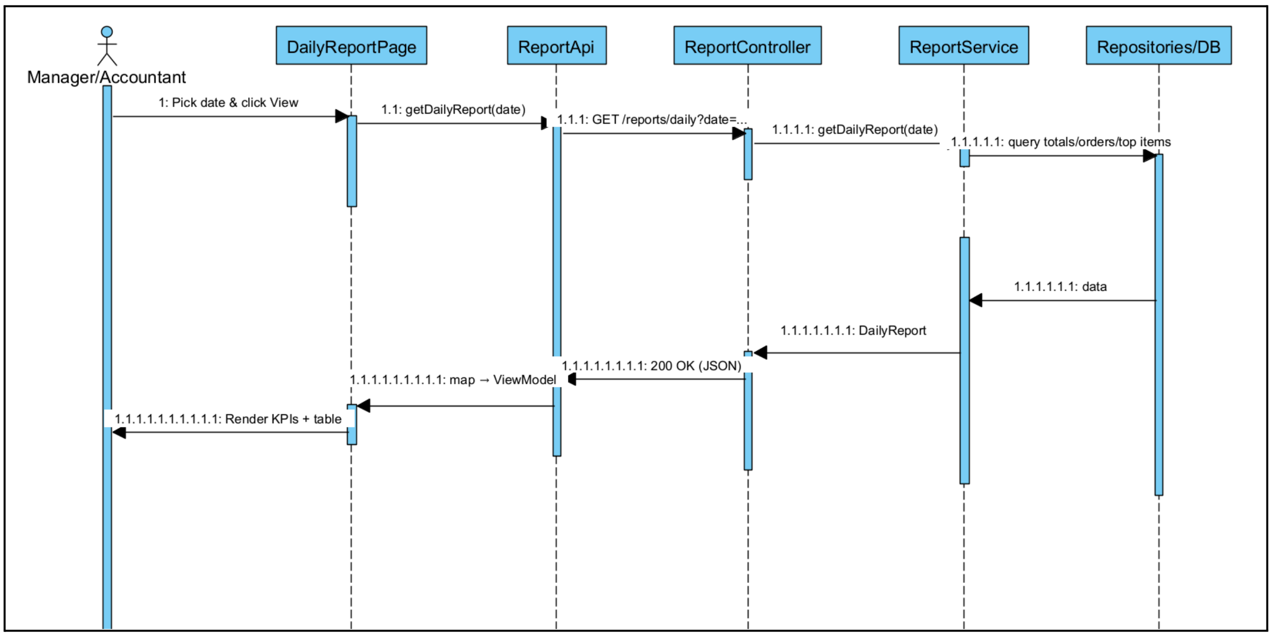
After entering or selecting the necessary information, the user clicks the “Add” button to record the task details.

If needed, the user can click “Save as CSV” to export the summarized work report for documentation or performance review.

No	Object/ control name	Type	Required	Length	Description
1	date	date picker	true	-	Select the specific date to view the daily report.
2	totalWorkingHours	text	false	-	Display total working hours

					for the selected date.
3	totalOvertimeHours	text	false	-	Show total overtime hours worked during the selected date.
4	completedTasks	text	false	-	Show the average order value (totalSales ÷ totalOrders).
5	inProgress	text	false	-	Display the number of tasks currently in progress.
6	issuesFound	text	false		Display the number of issues or problems encountered during the day.
7	completed	button	true		Show the number of completed tasks.
8	saveAsCsvButton	button	true		Export the displayed report to a downloadable CSV file.

1.6.4 Logic business process


Manager/Accountant
DailyReportPage
ReportApi
ReportController
ReportService
Repositories/DB
1: Pick date & click View
1.1: getDailyReport(date)
1.1.1: GET /reports/daily?date=...
1.1.1.1: getDailyReport(date)
1.1.1.1.1: query totals/orders/top items
1.1.1.1.1.1: data
1.1.1.1.1.1.1: DailyReport
1.1.1.1.1.1.1.1: 200 OK (JSON)
1.1.1.1.1.1.1.1.1: map → ViewModel
1.1.1.1.1.1.1.1.1.1: Render KPIs + table

2 DATABASE DESIGN

2.1 Entity Relationship Diagram

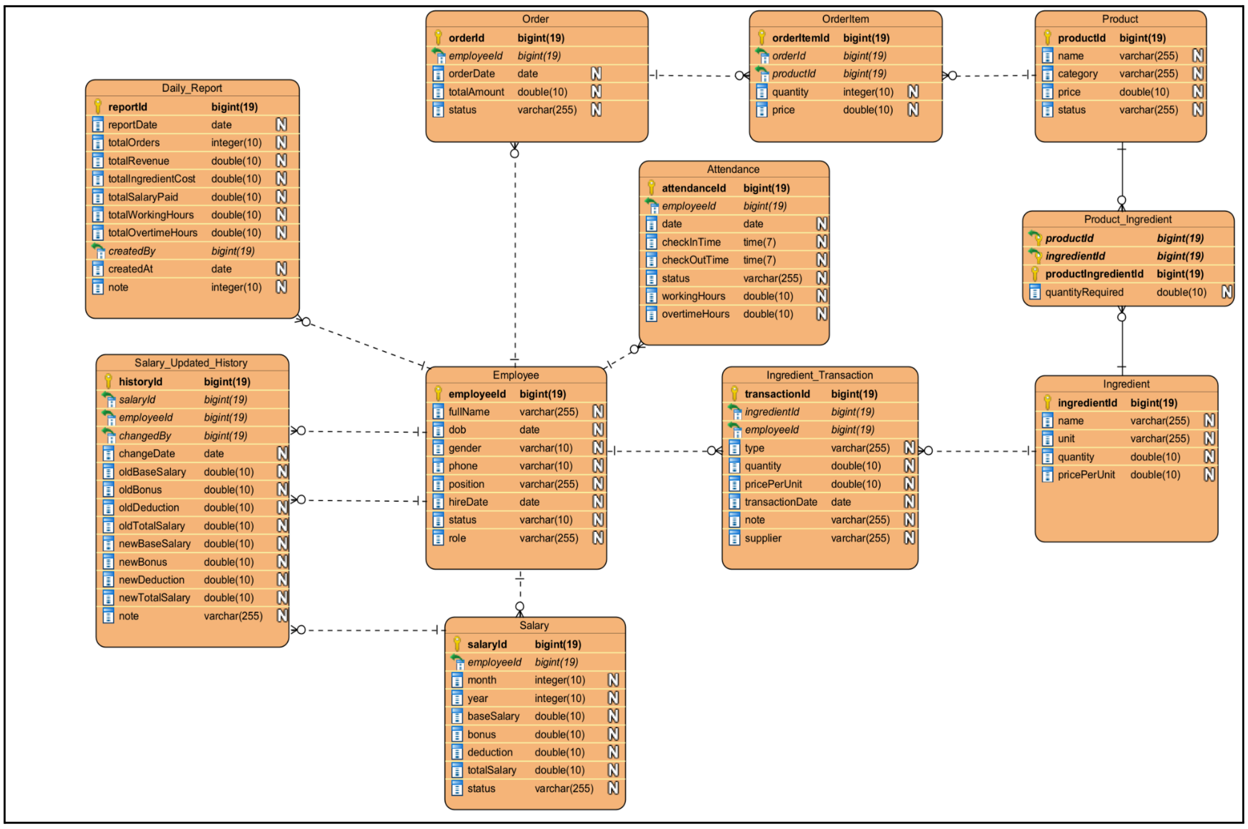


Figure: Entity Relationship Diagram for System

2.2 Database Diagram

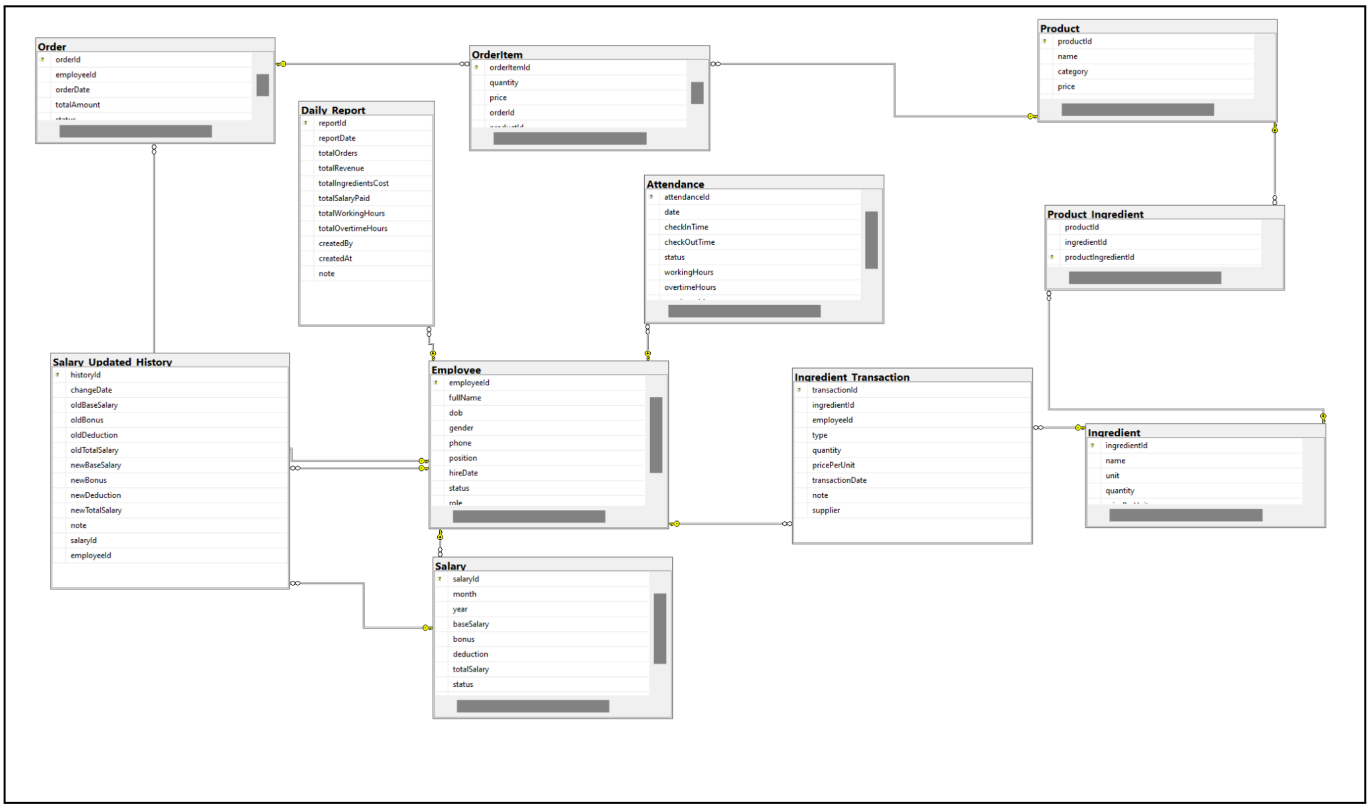


Figure 5: Database Diagram for System

2.3 Table Descriptions

2.3.1 Table Employee

No.	Attribute	Type	Constraints	Description
1	employeeId	bigint(19)	PK	Unique identifier for the employee
2	fullName	varchar(255)	NOT NULL	Full name of the employee
3	dob	date		Date of birth
4	gender	varchar(10)		Gender of the employee
5	phone	varchar(10)		Contact phone number
6	position	varchar(255)		Current job position
7	hireDate	date		Hiring date
8	status	varchar(255)		Employment status

9	role	varchar(255)		Role in the system (Admin, Staff, Manager)

2.3.2 Table Product

No.	Attribute	Type	Constraints	Description
1	productId	bigint(19)	PK	Unique identifier for the product
2	name	varchar(255)	NOT NULL	Product name
3	category	varchar(255)		Product category
4	price	double(10)		Price of the product
5	status	varchar(255)		Product status

2.3.3 Table Ingredient

No.	Attribute	Type	Constraints	Description
1	ingredientId	bigint(19)	PK	Unique identifier for the ingredient
2	name	varchar(255)	NOT NULL	Ingredient name
3	unit	varchar(255)		Measurement unit (kg, g, ml)
4	quantity	double(10)		Current stock quantity
5	pricePerUnit	double(10)		Price per measurement unit

2.3.4 Table Order

No.	Attribute	Type	Constraints	Description
1	orderId	bigint(19)	PK	Unique identifier for the order
2	employeeId	bigint(19)	FK → Employee (employeeId)	Employee who created the order
3	orderDate	date		Date when the order was placed
4	totalAmount	double(10)		Total amount of the order
5	status	varchar(255)		Order status

2.3.5 Table OrderItem

No.	Attribute	Type	Constraints	Description
1	orderItemId	bigint(19)	PK	Unique identifier for each order item
2	orderId	bigint(19)	FK → Order (orderId)	The order that this item belongs to
3	productId	bigint(19)	FK → Product (productId)	Product purchased in this order
4	quantity	integer(10)		Quantity ordered
5	price	double(10)		Price per unit at purchase time

2.3.6 Table Ingredient_Transaction

No.	Attribute	Type	Constraints	Description
1	transactionId	bigint(19)	PK	Unique identifier for each transaction
2	ingredientId	bigint(19)	FK → Ingredient (ingredientId)	Ingredient involved in the transaction
3	employeeId	bigint(19)	FK → Employee (employeeId)	Employee who processed the transaction
4	type	varchar(255)		Type of transaction (import/export)
5	quantity	double(10)		Quantity imported or exported
6	pricePerUnit	double(10)		Price per unit in the transaction
7	transactionDate	date		Date of the transaction
8	supplier	varchar(255)		Supplier name
9	note	varchar(255)		Additional notes

2.3.7 Table Attendance

No.	Attribute	Type	Constraints	Description

1	attendanceId	bigint(19)	PK	Unique identifier for each attendance record
2	employeeId	bigint(19)	FK → Employee (employeeId)	Employee who checked in
3	date	date		Attendance date
4	checkInTime	time(7)		Check-in time
5	checkOutTime	time(7)		Check-out time
6	status	varchar(255)		Attendance status (Present, Absent, Late)
7	workingHours	double(10)		Total working hours
8	overtimeHours	double(10)		Total overtime hours

2.3.8 Table Salary

No.	Attribute	Type	Constraints	Description
1	salaryId	bigint(19)	PK	Unique identifier for salary record
2	employeeId	bigint(19)	FK → Employee (employeeId)	Employee receiving the salary
3	month	integer(10)		Salary month
4	year	integer(10)		Salary year
5	baseSalary	double(10)		Base salary amount
6	bonus	double(10)		Bonus amount
7	deduction	double(10)		Deduction amount
8	totalSalary	double(10)		Total calculated salary
9	status	varchar(255)		Salary payment status

2.3.9 Table Salary_Updated_History

No.	Attribute	Type	Constraints	Description
1	historyId	bigint(19)	PK	Unique identifier for each salary update record

2	salaryId	bigint(19)	FK → Salary (salaryId)	Related salary record
3	employeeId	bigint(19)	FK → Employee (employeeId)	Employee whose salary was updated
4	changedBy	bigint(19)	FK → Employee (employeeId)	Employee who made the change
5	changeDate	date		Date of salary change
6	oldBaseSalary	double(10)		Previous base salary
7	oldBonus	double(10)		Previous bonus amount
8	oldDeduction	double(10)		Previous deduction amount
9	oldTotalSalary	varchar(255)		Previous total salary
10	newBaseSalary	double(10)		Updated base salary
11	newBonus	double(10)		Updated bonus amount
12	newDeduction	double(10)		Updated deduction amount
13	newTotalSalary	double(10)		Updated total salary
14	note	varchar(255)		Additional comments or reasons for the change

2.3.10 Table Product_Ingredient

No.	Attribute	Type	Constraints	Description
1	productId	bigint(19)	PK, FK → Product (productId)	Product associated with the ingredient
2	ingredientId	bigint(19)	PK, FK → Ingredient (ingredientId)	Ingredient used in the product
3	productIngredientId	bigint(19)	FK → Product (productId)	Unique relation identifier
4	quantityRequired	double(10)		Quantity of ingredient needed per product unit

2.3.11 Table Daily_Report

No.	Attribute	Type	Constraints	Description
1	reportId	bigint(19)	PK	Unique identifier of the report
2	reportDate	date		The date of the report
3	totalOrders	int		Total number of orders on that day
4	totalRevenue	double(10)		Total revenue from Order.totalAmount
5	totalIngredientsCost	double(10)		Total ingredient cost for that day
6	totalSalaryPaid	double(10)		Total salary paid on that day
7	totalWorkingHours	double(10)		Total working hours of all employees
8	totalOvertimeHours	double(10)		Total overtime hours of all employees
9	createdBy	bigint(19)	FK → Employee (employeeId)	Employee who created the report
10	createdAt	datetime		Date and time the report was created
11	note	varchar(255)		Optional notes