Deliverable 3

A. Report Requirements:

List of Functional and Non-Functional Requirements for Development Phase 1:

1. Initial Non-Functional Requirements:

1.1 **Initial Setup of Back-end repository structure**: Four layered architecture was used to implement back end for this project. Different Packages were created for each layer. The packages are application layer, config layer, controller layer, entities layer, exceptions layer, repository layer, service, and service implantation layer. Global configurations were stored in resources layer. Testing layer was also created for unit test code coverage for controller, entities, and service layers. Spring boot framework was used for entire back-end development.

1.2 **Initial Setup of Front-end repository structure**: Different directories are created for better code readability and maintainability. Reusable components are created in the components folder. Service integrations are done in the services folder. Different web pages are done in pages folder. React library, CSS and Bootstrap were used for the entire front-end development.

1.3 **Initial Database setup:** To store the data, we have used MySQL workbench. JPA framework was used which converts entities in java into tables in MySQL. Database configuration is given in resources folder in back end. MySQL workbench is installed in my local and created username and password. Database connection is successful, and we can view all the entities based on mapping between entities.

1.4 **Initial environment and Azure cloud setup**: We have created cloud setup by creating VM service. We have connected to the server through SSH and installed docker and git inside the VM. We have cloned the code repository into the VM and created docker images for both back end and front end. MySQL image is also created from the Docker hub. Three docker containers are created for front end, backend, and database respectively.

1.5 **Initial PayPal Payment Gateway developer account setup**: PayPal developer account is created with my mail and configured an application which has generated client key and client secret. PayPal related configuration was provided inside resources folder. The authentication was successful with PayPal gateway and we have created payment integrations.

1.6 **Initial Multi Factor Authentication related configurations setup**: OTP and user Id would be sent to user's email address. Email related configurations like email and password from the sender are provided in resources folder in back end.

1.7 **All software's installation related to back end and front end:** Eclipse, Postman, MySQL workbench and Java JDK are installed related to back end. Visual Studio code and Node JS are installed related to front end.

2. Back-end REST API Development Functional Requirements:

2.1 **Multi Factor Authentication Functionality**: The first step of authentication is to enter username and password. If the entered details are correct, the next step is to authenticate using OTP. The API is designed in such a way that username and password would be retrieved from front end and post HTTP request is triggered. Unique endpoint is exposed in Login controller layer. The username and password would be verified from database. If it is not successful, the back-end response would be 'User doesn't exist'. If it is successful, the 4-digit random OTP and expiration time would be generated and stored in OTP table. The OTP would be sent to user's email address. If the user enters the OTP before the expiration time the authentication was successful. If the OTP is expired, OTP would be stored in the database and sent again to user's email address.

2.**2 Forgot Password Functionality**: The API endpoint is unique and exposed in Login controller layer. The new password and username would be retrieved from front end. The new password would be stored in database to the respective user record by searching the user record by username. The password is also updated in customer and employee entity based on user role type.

2.3 **User Authorization:** Two types of users i.e., Customer and Employee were available in this application. The API endpoint is unique and exposed in Login controller layer. It is available in signup API. The user would be differentiated based on typeOfUser in user entity. If the user is of type 'customer' customer record would be created. When the forgot password functionality is triggered, the password would also get updated in the respective user record based on typeOfUser.

2.4 **Add Vehicle Information**: The API endpoint is unique and exposed in dashboard controller. The vehicle related details would be retrieved from front end. The endpoint also has the information of customer ID. The vehicle entity has the foreign key 'customer ID'. If the customer was found with the customer ID, the vehicle details would be stored in table with customer ID as the foreign key.

2.5 **Retrieval of All Products**: The API endpoint is unique and exposed in dashboard controller. The products would be retrieved from the products table. The products response is filtered with required details and back-end response is sent back to the customer.

2.6 **Retrieval of All Categories**: The API endpoint is unique and exposed in dashboard controller. The categories would be retrieved from the Product Category table. The category response is

filtered with required details and back-end response is sent back to the customer. One category contains multiple products. So, the mapping would be One to Many.

2.7 **Updating Personal Details**: The API endpoint is unique and exposed in customer controller. The updated customer details would be retrieved from front end. The endpoint also has the information of customer ID. The customer entity has the foreign key 'customer ID'. If the customer was found with the customer ID, the updated personal details would be stored in table.

2.8 **Nearest Store Locator**: The API endpoint is unique and exposed in dashboard controller. The username would be retrieved from front end and if the username was found in customer table, next sequential service call would be triggered which would fetch the nearest location details based on customer's zip code which would be retrieved from customer record. The required location details like address, contact Number, operating hours etc., would be sent as back-end response to the customer.

2.9 **Get All Product Details**: The API endpoint is unique and exposed in dashboard controller. The product ID would be retrieved from front end and service call would be triggered which would fetch the product details based on product ID. The required product details like ProductName, product description, product Image etc., would be sent as back-end response to the customer.

2.10 **Get All Products by Category**: The API endpoint is unique and exposed in dashboard controller. The category Name would be retrieved from front end and service call would be triggered which would fetch all the product details based on category ID. The list of product details would be sent as back-end response to the customer.

2.11 **Delivery Mode Selection**: There are two modes of selection i.e., Add to pick up and add to cart. If the user checks out products to cart, the order details would be sent to customer after completing payment. If the user opts for add to pick up, after completing payment user would be able to view online store details and employee details to pick the order.

2.12 **Adding Items to Cart**: The API endpoint is unique and exposed in customer controller. Once user add products to cart, the added product details would be retrieved from front end. If the combination of user ID and product ID does not exist in the cart entity, new entry would be persisted into table. If it exists, the product quantity would be increased by 1.

2.13 **Removal of Items from Cart**: The API endpoint is unique and exposed in customer controller. If the combination of user ID and product ID exist in the cart entity, the cart item would be deleted from cart entity.

2.14 **Completing Payment**: The API endpoint is unique and exposed in payment Info controller. The user ID and cart details would be retrieved from front end and the sequential service call would be triggered which would initiate payment with the total amount. It would give PayPal link as back-end response. If the user completes the payment successfully, the amount will get deducted from PayPal balance and user would be redirected to success payment screen which is sent as

back-end response and the payment record is also stored in payment Info entity with payment status as 'SUCCESS'.

2.15 **Cancelling Payment**: The API endpoint is unique and exposed in payment Info controller. The user ID and cart details would be retrieved from front end and the sequential service call would be triggered which would initiate payment with the total amount. If the user cancels the payment, the user would be redirected back to cart screen which is sent as back-end response and the payment record is also stored in payment Info entity with payment status as 'Cancelled'.

3. Functional Requirements for Designing Front end UI screens and Integration with Back-end API:

3.1 **Forgot Password Functionality**: Forgot Password page is created in pages folder and route was added in main App folder. The page was designed with CSS and bootstrap for better usability. If the user clicks on forgot password button in login page, we will land on forgot password screen . The screen was designed with two text boxes one with username and the other with password. The entered details would be stored in useState hook and service call would be triggered which gets integrated with axios library to hit back-end API. If the status is '200' the user would be redirected back to login screen.

3.2 **Multi Factor Authentication Functionality**: Login page and sign-up pages are created in pages folder and routes were added in main App folder. The pages are designed with CSS and bootstrap for better usability. The login and sign-up actions would be available in navbar. If the user clicks on sign-up button, the user would be redirected to sign-up screen where user would fill all the personal details. The entered details would be stored in useState hook and service call would be triggered which gets integrated with axios library to hit back-end API. If the status is '200' the user would be redirected back to login screen to login into the application. If the user clicks on login button, the user would be redirected to login screen where user would fill username and password. The entered details would be stored in useState hook and service call would be triggered which gets integrated with axios library to hit back-end API. If the status is '200' the OTP popup would be opened and user enters the generated OTP which was sent to the customer's mail. If the entered OTP is valid and correct, user would be redirected to dashboard screen.

3.**3 Add Vehicle Information**: Add Vehicle Details page is created in pages folder and route was added in main App folder. The page was designed with CSS and bootstrap for better usability. If the user clicks on add vehicle button in dashboard page, we will land on vehicle details screen . The screen was designed with multiple text boxes with all the required fields. The entered details would be stored in useState hook and service call would be triggered which gets integrated with axios library to hit back-end API. If the status is '200' the user would be redirected back to dashboard screen.

3.4 **Retrieval of All Products**: The dashboard screen has the products component which would be of carousel design. The page was designed with CSS and bootstrap for better usability. The list of products would be fetched from back end and it would be rendered asynchronously on dashboard.

3.5 **Retrieval of All Categories**: The dashboard screen has the category component which would be of carousel design. The page was designed with CSS and bootstrap for better usability. The list of categories would be fetched from back end and it would be rendered asynchronously on dashboard.

3.6 **Updating Personal Details**: Update personal details page is created in pages folder and route was added in main App folder. The page was designed with CSS and bootstrap for better usability. If the user clicks on update personal details button in dashboard page, we will land on personal details screen . The screen was designed with multiple text boxes with all the required fields. The entered details would be stored in useState hook and service call would be triggered which gets integrated with axios library to hit back-end API. If the status is '200' the user would be redirected back to dashboard screen.

3.7 **Nearest Store Locator**: The dashboard screen has the location details in the navbar. The navbar was designed with CSS and bootstrap for better usability. The nearest location details would be fetched from back end and it would be rendered asynchronously on dashboard.

3.8 **Get All Product Details**: If the user clicks on any of the product in the dashboard, the user would be landed on product details page and routes are provided in main App folder. The page was designed with CSS and bootstrap for better usability. The screen would contain information of the selected product and it contains 'Add to pick up' and 'Add to cart' buttons.

3.9 **Get All Products by Category**: If the user clicks on any of the category in the dashboard, the user would be landed on product by category page and routes are provided in main App folder. The page was designed with CSS and bootstrap for better usability. The screen would contain information of the list of products based on category and every product contains 'Add to pick up' and 'Add to cart' buttons.

3.**10 Delivery Mode Selection**: User has the option of selecting Add to pick up' and 'Add to cart' buttons in product details page and products by category page. 'Add to pick up' option is also available in carts page beside each cart item.

3.11 **Adding Items to Cart**: Cart Component is created in pages folder and route was added in main App folder. The page was designed with CSS and bootstrap for better usability. If the user clicks on view cart button in navbar, we will land on cart details screen. The screen was designed in such a way that it would display list of product items in cart page. The order summary would be displayed on the right with total amount and 'Complete Payment through PayPal' button. Every item in cart page has information of product, increase, decrease and delete button.

3.12 **Removal of Items from Cart**: Every item in cart page has delete button. If the user clicks on delete button it would remove the item from the cart screen.

3.**13 Completing Payment**: Order Details Component is created in pages folder and route was added in main App folder. The page was designed with CSS and bootstrap for better usability. If the user clicks on 'Complete Payment through PayPal' button in cart page , we will land on PayPal payments screen. The user completes the payment and if it is successful, user would be redirected to order details screen and it displays payment ID and status of payment.

3.14 **Cancelling Payment**: If the user clicks on 'Complete Payment through PayPal' button in cart page, we will land on PayPal payments screen. If the user cancels the payment, user would be redirected back to cart page.

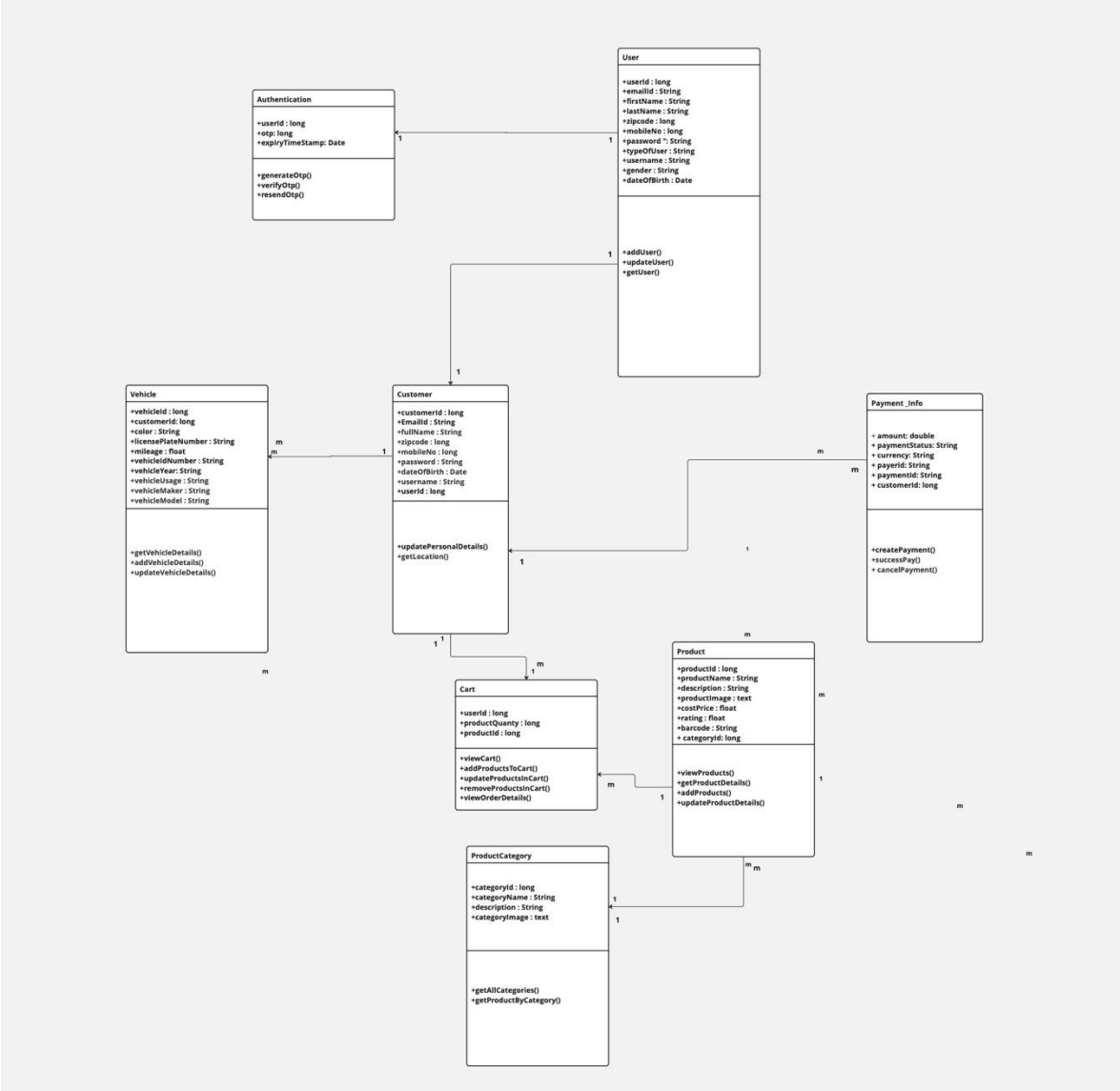
4**. Other Non-Functional Requirements**:

4.1 **Code reviews and merge requests for phase 1 developed code**: The code reviews were performed and code was pushed by all the developers and it was merged in main branch.

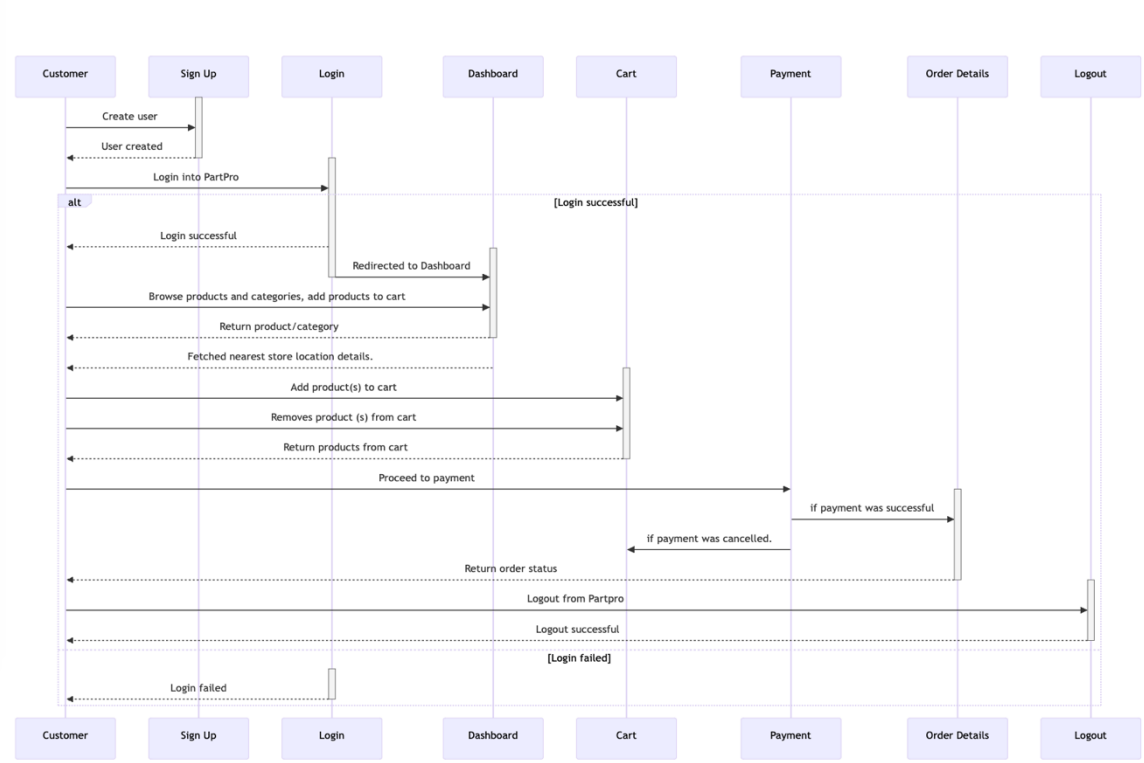
4.2 **Code deployment for phase 1 developed code**: The updated code would be retrieved from main branch and the latest code would be pushed to azure cloud VM. The docker images would be updated and the containers are up and running with the latest code.

B. **Report UML**:

1. **Class Diagram:** This development phase includes important entities and core features are implemented in this phase. The entities are Authentication, User, Customer, Vehicle, Payment_Info, Cart, Product and Product Category. The user can be a of type customer and customer is part of a user. So, the mapping between customer and user is One to One bidirectional. The mapping between authentication and customer is One to One as we would store OTP of every user. One customer can own multiple vehicles. So, the mapping between customer and vehicle is One to many. One customer can perform multiple payments. So, the mapping between customer and payments is One to many. One customer can add multiple items into cart. So, the mapping between customer and cart is One to Many. One product can be part of multiple items in cart. So, the mapping between product and cart is One to Many. A category contains multiple products. So, the mapping between category and products is One to Many.

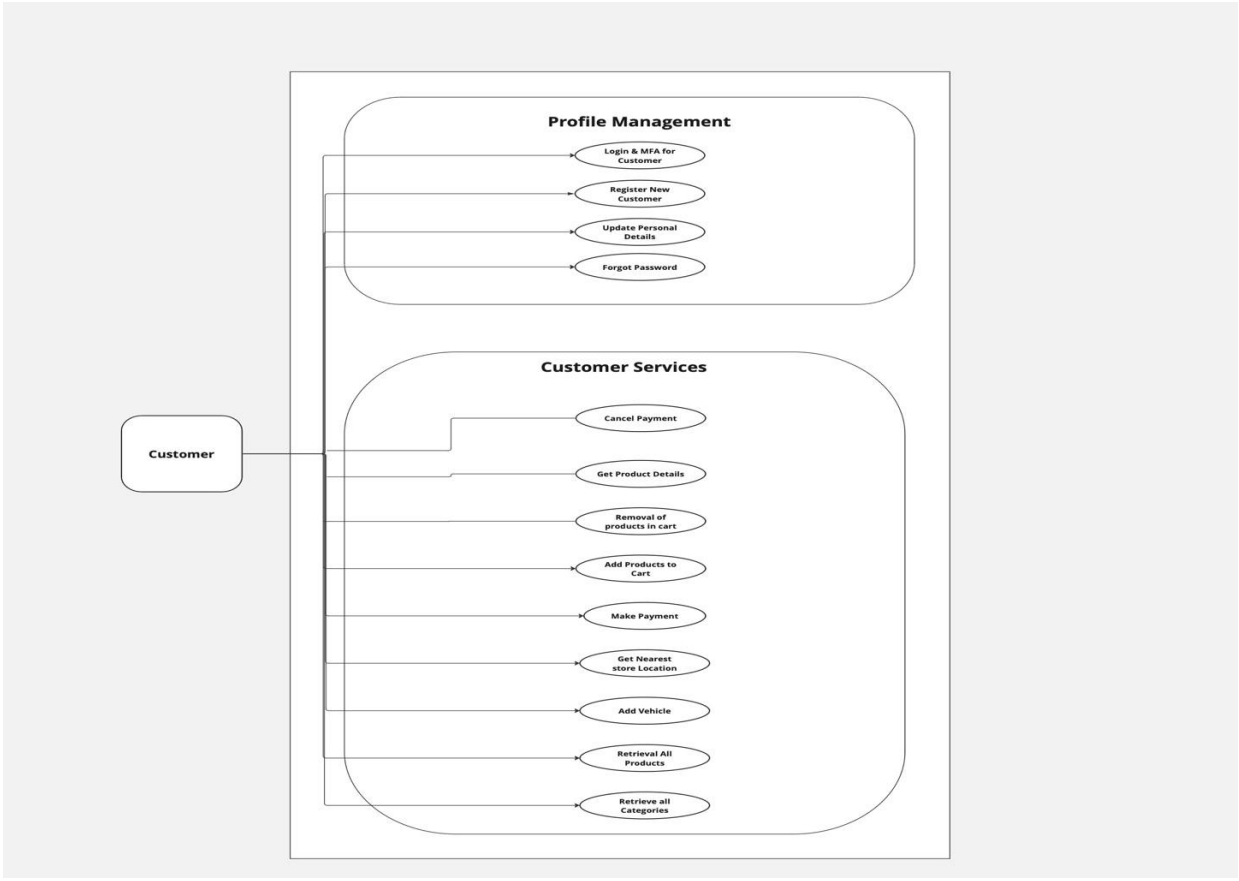


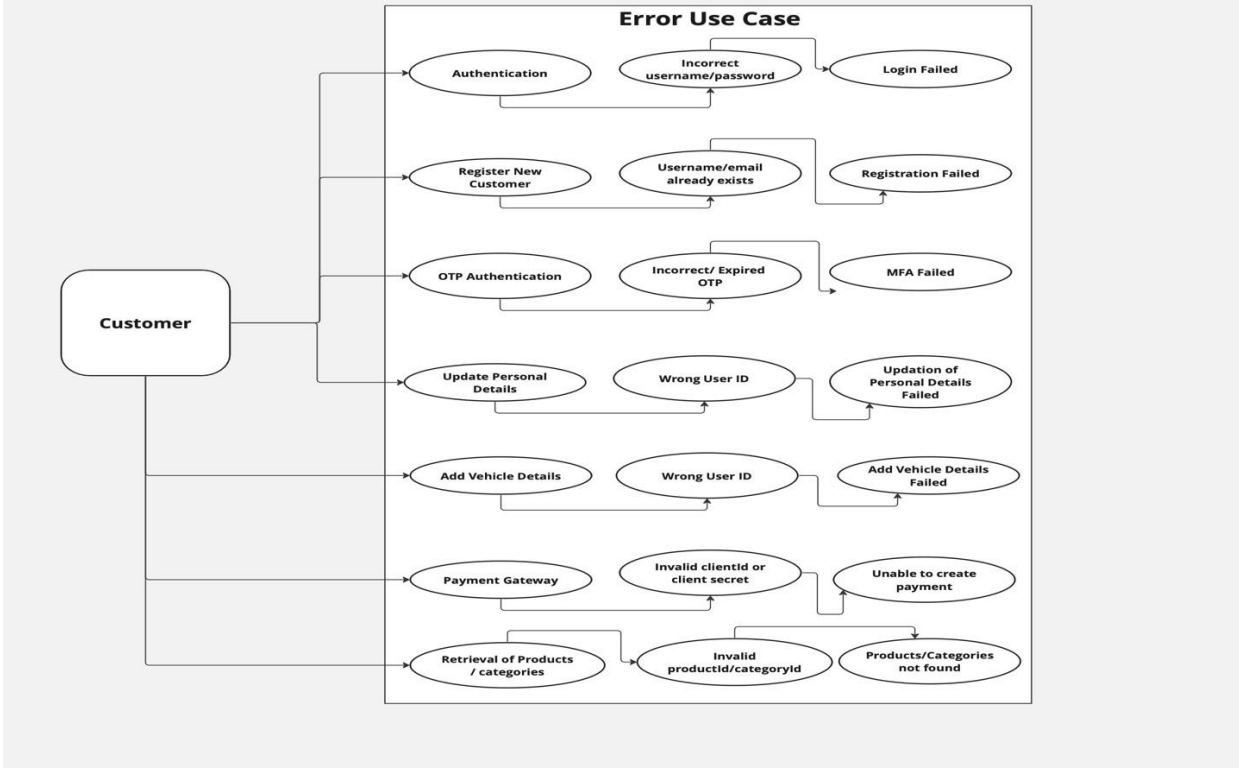
2. **Sequence Diagram**: This phase of development concentrates mainly on customer sequences with the application. Customer can register, login and logout from the application. If the login is successful, customer would land on dashboard. Customer can view multiple products and categories on dashboard. Customer can add items to cart and could also remove product from cart. Customer can also complete and cancel payment. If the payment is successful, user would land on order details page. If the payment is cancelled, user would navigate back to cart page.



3. **<u>Use Case Diagram</u>**: Th actor would be customer for this development phase. The actions include Login, MFA, register new customer, update personal details, forgot password. The other customer services include cancel payment, get product details, Removal of products in cart, add products to cart, make payment, get nearest location details, add vehicle, retrieve all products, and retrieve all categories.

   The negative use case diagram includes customer as an actor. If the customer provides incorrect username/ password, the login would be failed. If the username/email already exists , the registration to the application would be failed. If the OTP is expired or incorrect, Multi factor authentication would be failed. If the user ID is wrong, customer cannot update personal details and cannot add vehicle details. If the client ID or client secret for PayPal payment gateway are invalid, customer is unable to create payment. If the product ID/ category ID is invalid , customer cannot find product/ category details/


Profile Management
Login & MFA for Customer
Register New Customer
Update Personal Details
Forgot Password
Customer Services
Customer
Cancel Payment
Get Product Details
Removal of products in cart
Add Products to Cart
Make Payment
Get Nearest store Location
Add Vehicle
Retrieval All Products
Retrieve all Categories


Error Use Case
Customer
Authentication
Incorrect username/password
Login Failed
Register New Customer
Username/email already exists
Registration Failed
OTP Authentication
Incorrect/ Expired OTP
MFA Failed
Update Personal Details
Wrong User ID
Updation of Personal Details Failed
Add Vehicle Details
Wrong User ID
Add Vehicle Details Failed
Payment Gateway
Invalid clientId or client secret
Unable to create payment
Retrieval of Products / categories
Invalid productId/categoryId
Products/Categories not found

C. **Report Test Cases**: The main core functionalities were tested in this phase. Unit Tests were written for all major functionalities. Junit and Mockito frameworks are used to write unit test cases.

Junit: The Java programming language has an open-source framework for unit testing called JUnit. Test runners are used to execute tests, assertions are used to test expected results, and annotations are used to identify test methods in JUnit.

Mockito Framework: It enables developers to set up and construct fake objects to test a class's behavior independently, particularly in cases when the class has external dependencies.

I have written multiple test cases for code coverage but sharing major unit test cases here. The tests are in multiple folders. More unit tests are written to have better code quality and code coverage.

partpro-managemensystem/src/backend/src/test/java/com/unt/se/ppms/controller
partpro-managemensystem/src/backend/src/test/java/com/unt/se/ppms/dto
partpro-managemensystem/src/backend/src/test/java/com/unt/se/ppms/entities
partpro-managemensystem/src/backend/src/test/java/com/unt/se/ppms/service/impl.

But the major core unit tests would be available in Controller layer which mocks the requests and gives the stubbed response.

Test Case 1: 'testLoginFirstStep':

Purpose: The goal is to confirm that a user login attempt is handled appropriately by the LoginController class's loginFirstStep method.

Functionality Under Test: This technique examines the user authentication process that occurs during the first login stage using a username and password.

Test Steps:

Arrange (Setup the Test Environment): A 'userData' object is created and populated with sample data, representing a mock user's data.

When called with any string arguments for username and password, the loginService.loginUserFirstStep() method is mocked using Mockito to return the preconfigured UserData object.

Act(Execute the Operation): Using the endpoint /ppms/user/login/{username}/{password} with the sample login credentials "janedoe" and "jan1234", a POST request is created.

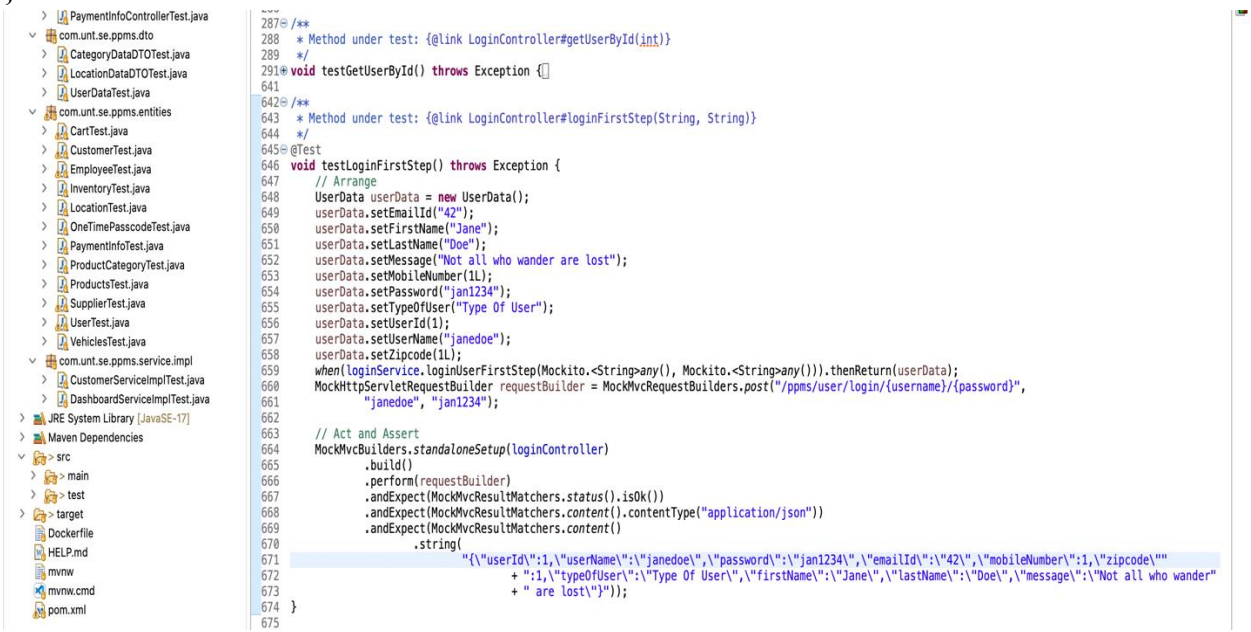
Using the MockMvc framework, which mimics MVC framework behavior without launching an actual HTTP server, the request is delivered to the loginController.

Assert(Verify the Outcome): The test claims that the login attempt was successfully handled because the HTTP response status is 200 OK.

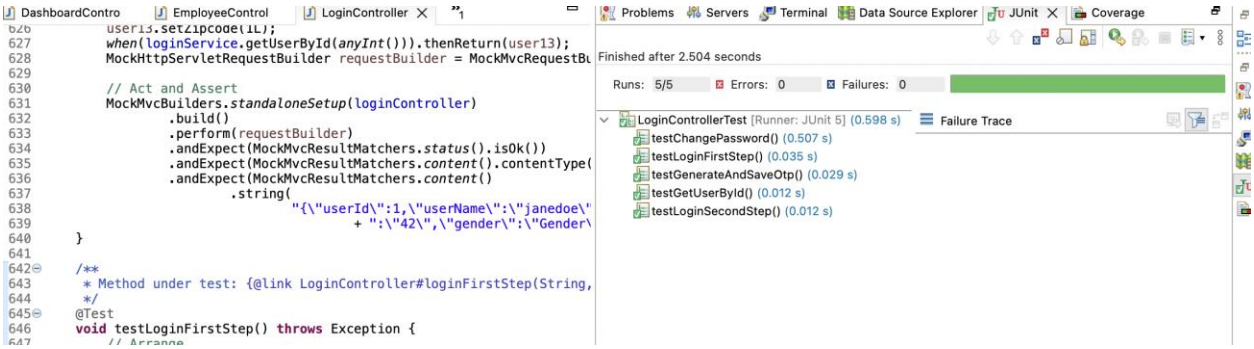
It confirms that, as would be expected for a JSON response, the response content type is application/json.

To guarantee that the correct user data is returned upon a successful login, it lastly verifies that the content of the response precisely matches the JSON representation of the preset UserData object.

Inputs:
'userName': "jandoe"
'password': "jan1234"
Expected Outputs:
HTTP Status: '200 OK'
Content Type: 'application/json'
JSON Content: {
"userId": 1,
"userName": "janedoe",
"password": "jan1234",
"emailId": "42",
"mobileNumber": 1,
"zipcode": 1,
"typeOfUser": "Type Of User",
"firstName": "Jane",
"lastName": "Doe",
"message": "Not all who wander are lost"
}



The above test case is' TestLoginFirstStep' in LoginControllerTest. So, I have run LoginControllerTest and the above test result is passed and shown in the below image.



Test Case 2: 'testLoginSecondStep'

Purpose: This test is designed to verify that the LoginController is handling the OTP (One-Time Password) verification step, which is the second stage in the login process.

Functionality Under Test: The method's ability to complete verification and deliver the desired result is confirmed when the right user ID and OTP are supplied.

Test Steps:

Arrange (Setup the Test Environment): Any integer (representing user ID) and any long (representing OTP) can be used to fake the loginService.loginUserSecondStep() method, which returns the string "Login User Second Step" when invoked.

Act(Execute the Operation): Mock user ID 1 and OTP 1L are used to make a POST request to the endpoint /ppms/user/login/{id}/verifyotp/{otp}.

MockMvc simulates the functionality of the MVC framework in an isolated environment and sends this request to the loginController.

Assert(Verify the Outcome): The request was successful, since the return status of 200 OK is confirmed.

The response is expected to have the content type "text/plain;charset=ISO-8859-1".

The string "Login User Second Step" is expected to appear in the response body, indicating that the controller has successfully completed the OTP verification process and returned the expected response text.

Inputs: User ID: 1

OTP: 1L

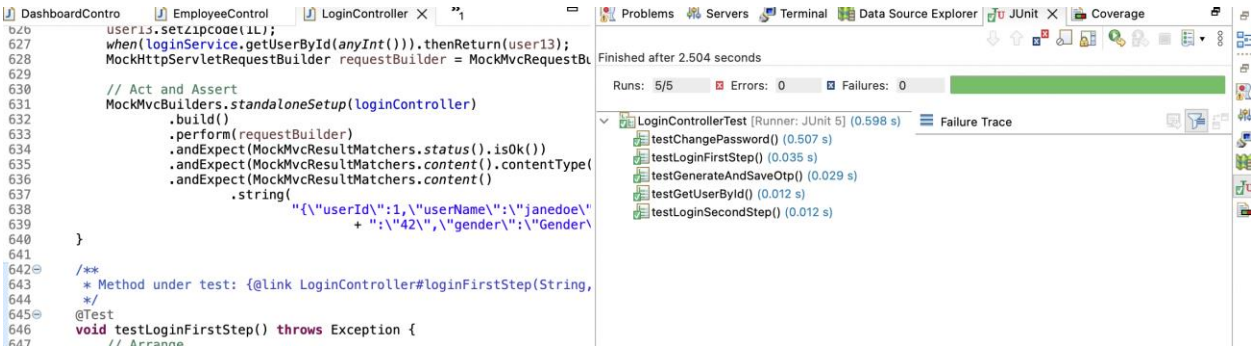
Expected Outputs:

HTTP Status: 200 OK

Content Type: text/plain;charset=ISO-8859-1

Response Body: "Login User Second Step"

The above test case 'TestLoginSecondStep' is in LoginControllerTest. So, I have run LoginControllerTest and the above test result is passed and shown in the below image.



Test Case 3: 'testGetNearestLocation':

Purpose: The goal is to make sure that the dashboardController provides accurate data in JSON format and that the getNearestLocation method in the DashboardService appropriately retrieves location information based on a provided username.

Functionality Under Test: With the help of a test, a user called "foo" can retrieve the closest location information. It assesses the capacity of the controller layer to process the request and provide the data in the appropriate format, as well as the ability of the service layer to return accurate data.

Test Steps:

Arrange (Setup the Test Environment): Sample location information is constructed and loaded into a LocationDataDTO object, including the address, phone number, location ID, operation hours, review, and zip code.

To simulate the retrieval of location data for any username, the dashboardService.getNearestLocation() method is mocked to return the LocationDataDTO object when invoked with any string parameter.

Create a GET request with the request parameter username set to "foo" and send it to the endpoint /ppms/dashboard/getLocation.

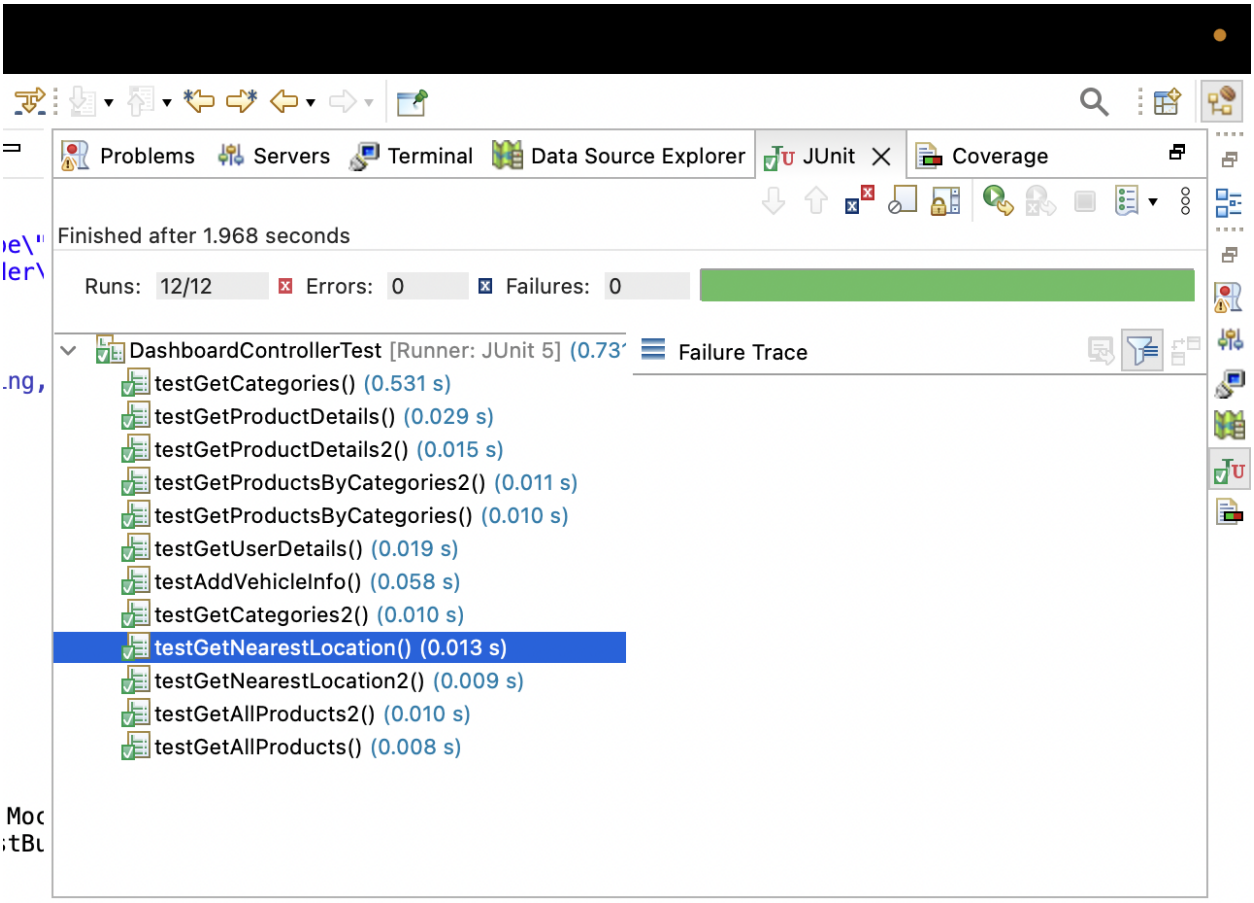
Act(Execute the Operation): Using the MockMvc framework, a simulation of the MVC environment, the GET request is made to the dashboardController.
Assert(Verify the Outcome): The test claims that it was successfully handled because the HTTP response status is 200 OK.
It confirms that, as would be expected for a JSON response, the response content type is application/json.
To guarantee that the correct location data is returned, it lastly verifies that the content of the response precisely matches the JSON representation of the preset LocationDataDTO object.

Inputs: Username parameter: "foo"
Expected Outputs: HTTP Status: 200 OK
Content Type: application/json
JSON Content: {
"locationId": 1,
"contactNumber": 1,
"review": "Review",
"operatingHrs": "Operating Hrs",
"address": "42 Main St",
"zipcode": 1
}

```
/**
 * Method under test: {@link DashboardController#getNearestLocation(String)}
 */
@Test
void testGetNearestLocation() throws Exception {
    // Arrange
    LocationDataDTO locationDataDTO = new LocationDataDTO();
    locationDataDTO.setAddress("42 Main St");
    locationDataDTO.setContactNumber(1L);
    locationDataDTO.setLocationId(1L);
    locationDataDTO.setOperatingHrs("Operating Hrs");
    locationDataDTO.setReview("Review");
    locationDataDTO.setZipcode(1L);
    when(dashboardService.getNearestLocation(Mockito.<String>any())).thenReturn(locationDataDTO);
    MockHttpServletRequestBuilder requestBuilder = MockMvcRequestBuilders.get("/ppms/dashboard/getLocation")
            .param("username", "foo");

    // Act and Assert
    MockMvcBuilders.standaloneSetup(dashboardController)
            .build()
            .perform(requestBuilder)
            .andExpect(MockMvcResultMatchers.status().isOk())
            .andExpect(MockMvcResultMatchers.content().contentType("application/json"))
            .andExpect(MockMvcResultMatchers.content()
                    .string(
                            "{\"locationId\":1,\"contactNumber\":1,\"review\":\"Review\",\"operatingHrs\":\"Operating Hrs\",\"address\":\"42 Main"
                                    + " St\",\"zipcode\":1}"));
}
```

The above test case ‘testGetNearestLocation’ is in DashboardControllerTest. So, I have run DshboardControllerTest and the above test result is passed and shown in the below image.



Test Case 4: 'testGetCategories():

Purpose: The goal is to make sure that the dashboardController provides accurate data in JSON format and that the getAllCategories() method in the DashboardService appropriately retrieves category information.

Functionality Under Test: It verifies that the controller layer can process the GET request and return the data as JSON, as well as that the service layer has successfully retrieved all category data.

Test Steps:

Arrange (Setup the Test Environment): When called, the dashboardService.getAllCategories() method returns a faked list of categories. For the endpoint /ppms/dashboard/getCategories, a GET request builder is made.

Act(Execute the Operation): Using MockMvc, the dashboardController executes the GET request, mocking the behavior of a client requesting the category list.

Assert(Verify the Outcome): The test claims that it was successfully handled because the HTTP response status is 200 OK.

It confirms that, as would be expected for a JSON response, the response content type is application/json.

To guarantee that the correct category data is returned, it lastly verifies that the content of the response precisely matches the JSON representation of the preset CategoryDataDTO object.

Inputs: No input parameters are specified directly; the test simply triggers the request to get all categories. But here, we mock the response of categories list and create an

Expected Outputs: HTTP Status: 200 OK

Content Type: application/json

JSON Content: [

{

"categoryId": 1,

"categoryName": "brakes",

"categoryImage": "https://www.brakes.jpeg",

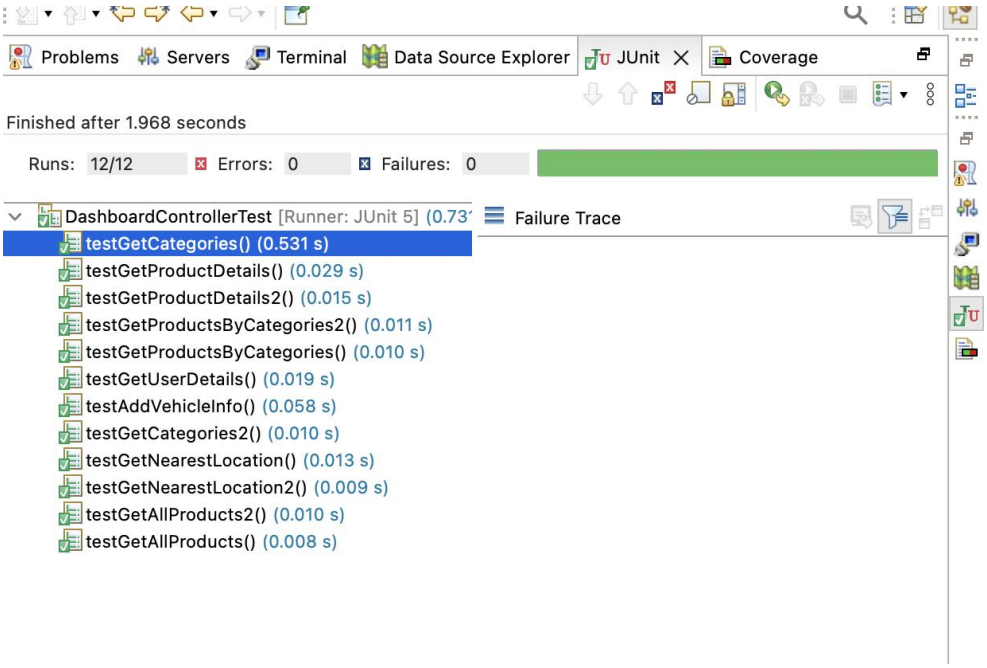
"description": "used for stopping vehicles"

}

]

```
693    /**
694     * Method under test: {@link DashboardController#getCategories()}
695     */
696    @Test
697    void testGetCategories() throws Exception {
698        // Arrange
699        CategoryDataDTO categories=new CategoryDataDTO();
700        categories.setCategoryId(1);
701        categories.setDescription("used for stopping vehicles");
702        categories.setCategoryImage("https://www.brakes.jpeg");
703        categories.setCategoryName("brakes");
704        List< CategoryDataDTO> ca= new ArrayList<CategoryDataDTO>();
705        ca.add(categories);
706        when(dashboardService.getAllCategories()).thenReturn(ca);
707        MockHttpServletRequestBuilder requestBuilder = MockMvcRequestBuilders.get("/ppms/dashboard/getCategories");
708
709        // Act and Assert
710        MockMvcBuilders.standaloneSetup(dashboardController)
711                .build()
712                .perform(requestBuilder)
713                .andExpect(MockMvcResultMatchers.status().isOk())
714                .andExpect(MockMvcResultMatchers.content().contentType("application/json"))
715                .andExpect(MockMvcResultMatchers.content().string("[{\"categoryId\":1,\"categoryName\":\"brakes\",\"categoryImage\":\"https://www.brakes.jpeg\",\"description\":\"used for stopping vehicles\"
716    }
717
```

The above test case ‘testGetCategories’ is in DashboardControllerTest. So, I have run DshboardControllerTest and the above test result is passed and shown in the below image.



Test Case_5: testGetProductDetails()

Purpose: The goal is to make sure that the dashboardController provides accurate data in JSON format and that the getProductDetails() method in the DashboardService appropriately retrieves product information.

Functionality Under Test: The test case concentrates on the controller's ability to return this data in JSON format upon receiving a product ID from the service and on the service's capability to get product details.

Test Steps:

Arrange (Setup the Test Environment): When calling the dashboardService.getProductDetails(anyLong()) function with any long number as the product ID, the method is mocked to return the Products object.

With the endpoint /ppms/dashboard/getProductDetails and a parameter productId set to 1L, a GET request builder is set up.

Act(Execute the Operation): Using MockMvc, which simulates the web context without launching a real server, the defined GET request is forwarded to the dashboardController.

Assert(Verify the Outcome): The test claims that it was successfully handled because the HTTP response status is 200 OK.

It confirms that, as would be expected for a JSON response, the response content type is application/json.

The response's JSON content is verified to match the JSON string that should be returned, representing the collection of product data found in the Products object.

Inputs:

productId: The ID of the product to retrieve details for, set to 1L in this test.

Expected Outputs: HTTP Status: 200 OK

Content Type: application/json

JSON Content: {

"productId": 1,

"productName": "Product Name",

"productPrice": 10.0,

"productImage": "Product Image",

"barCode": "Bar Code",

"productDescription": "Product Description"

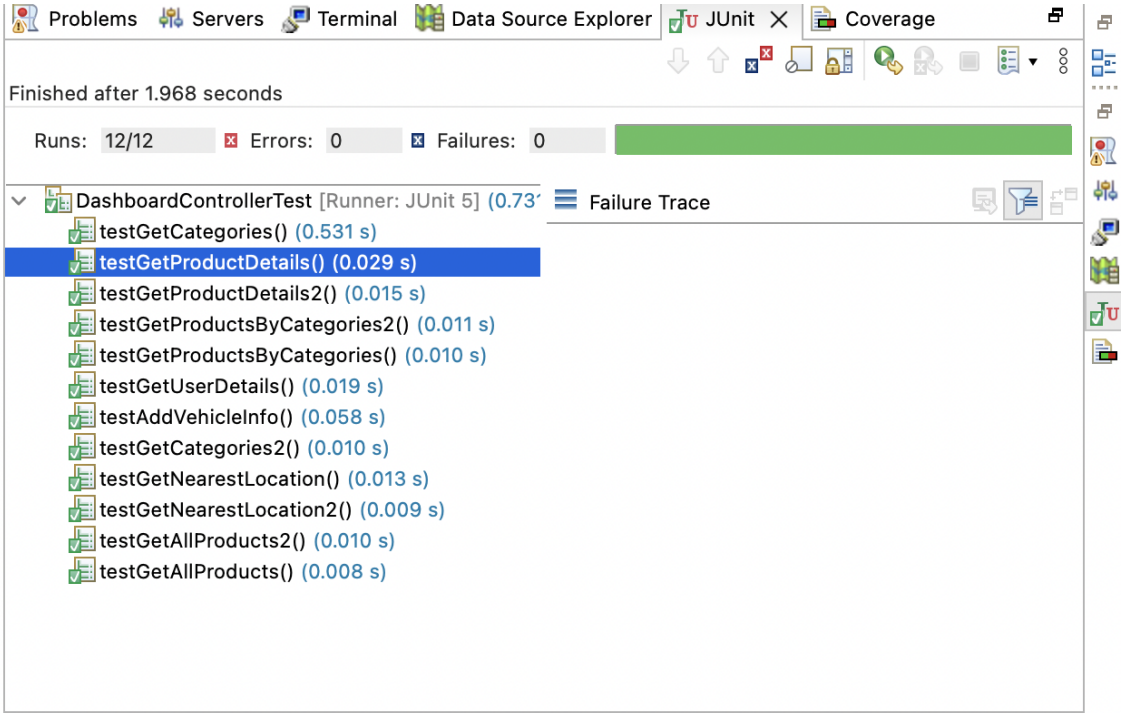
}

```
    /**
     * Method under test: {@link DashboardController#getProductDetails(long)}
     */
    @Test
    void testGetProductDetails() throws Exception {
        // Arrange
        ProductCategory category = new ProductCategory();
        category.setCategoryId(1L);
        category.setCategoryImage("Category Image");
        category.setCategoryName("Category Name");
        category.setDescription("The characteristics of someone or something");
        category.setProducts(new ArrayList<>());

        Products products = new Products();
        products.setBarCode("Bar Code");
        products.setCategory(category);
        products.setInventories(new ArrayList<>());
        products.setOnlineSales(new ArrayList<>());
        products.setProductDescription("Product Description");
        products.setProductId(1L);
        products.setProductImage("Product Image");
        products.setProductName("Product Name");
        products.setProductPrice(10.0f);
        products.setSuppliers(new HashSet<>());
        when(dashboardService.getProductDetails(anyLong())).thenReturn(products);
        MockHttpServletRequestBuilder getResult = MockMvcRequestBuilders.get("/ppms/dashboard/getProductDetails");
        MockHttpServletRequestBuilder requestBuilder = getResult.param("productId", String.valueOf(1L));

        // Act and Assert
        MockMvcBuilders.standaloneSetup(dashboardController)
                .build()
                .perform(requestBuilder)
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.content().contentType("application/json"))
                .andExpect(MockMvcResultMatchers.content()
                        .string(
                                "{\"productId\":1,\"productName\":\"Product Name\",\"productPrice\":10.0,\"productImage\":\"Product Image\",\"barCode\":
                                        + " Code\",\"productDescription\":\"Product Description\"}"));
    }
```

The above test case ‘testGetProductDetails’ is in DashboardControllerTest. So, I have run DshboardControllerTest and the above test result is passed and shown in the below image.



Test Case 6: 'testCancelPayment'

Purpose: The goal is to make sure that the PaymentInfoController provides accurate status code and redirection URL upon cancelling the payment.

Functionality Under Test: The goal of the test is to verify that the user is sent to the cart page and that the associated payment information is updated and retrieved upon request for a payment cancellation.

Test Steps:

Arrange (Setup the Test Environment): A GET request is configured to the endpoint /ppms/payment/{userId}/cancel with a path variable userId set to 1L and a request parameter paymentId set to "foo".

Act(Execute the Operation): Using the MockMvc framework, the paymentInfoController is subjected to the GET request..

Assert(Verify the Outcome): The redirected URL "http://localhost:3000/cart" is verified, suggesting that the user should be returned to their cart following the cancelation of their purchase.

Inputs: simulated values foe testing purpose

userId: 1L

paymentId: "foo"

Expected Outputs:

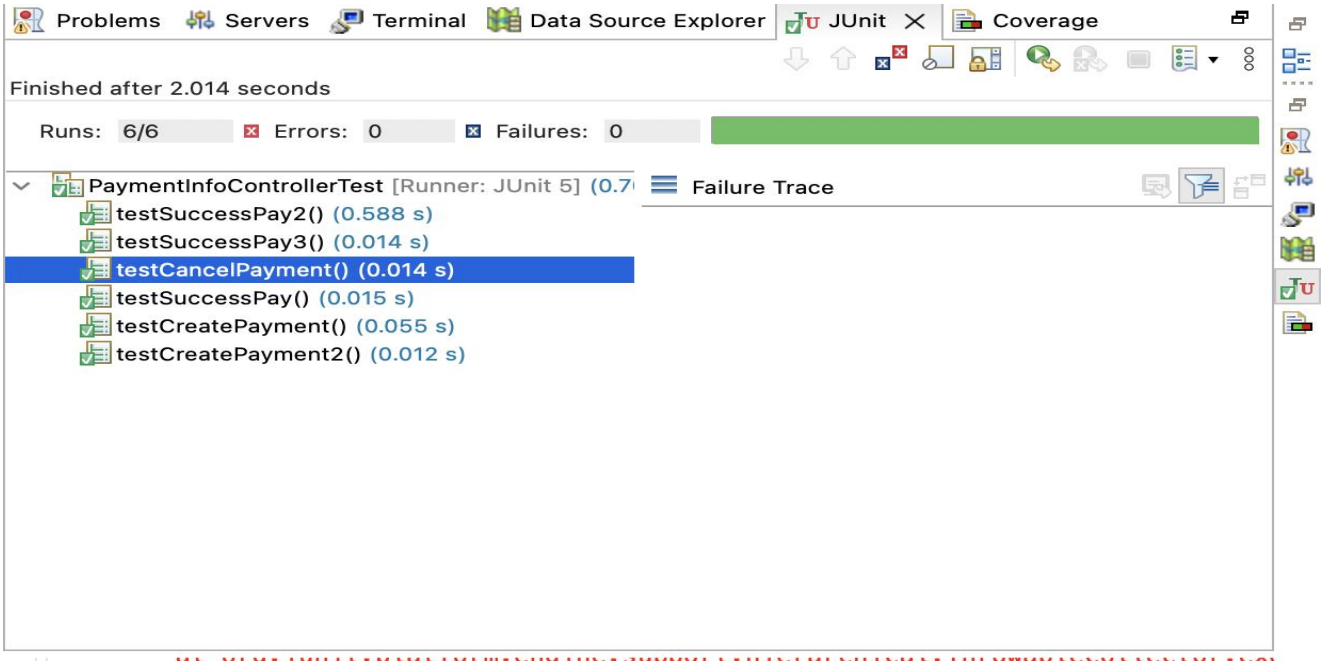
HTTP Status: 302 Found

Redirection URL: http://localhost:3000/cart

```
/**
 * Method under test: {@link PaymentInfoController#cancelPayment(String, long)}
 */
@Test
void testCancelPayment() throws Exception {
    // Arrange
    PaymentInfo paymentInfo = new PaymentInfo();
    paymentInfo.setAmount(10.0d);
    paymentInfo.setCurrency("GBP");
    paymentInfo.setPayerId("42");
    paymentInfo.setPaymentId("42");
    paymentInfo.setPaymentStatus(PaymentInfo.PaymentStatus.INITIAL.getStr());
    when(paymentInfoService.addOrUpdatePayment(Mockito.<PaymentInfo>any())).thenReturn("2020-03-01");
    when(paymentInfoService.getByPaymentId(Mockito.<String>any())).thenReturn(paymentInfo);
    MockHttpServletRequestBuilder requestBuilder = MockMvcRequestBuilders.get("/ppms/payment/{userId}/cancel", 1L)
            .param("paymentId", "foo");

    // Act and Assert
    MockMvcBuilders.standaloneSetup(paymentInfoController)
            .build()
            .perform(requestBuilder)
            .andExpect(MockMvcResultMatchers.status().isFound())
            .andExpect(MockMvcResultMatchers.redirectedUrl("http://localhost:3000/cart"));
}
```

The above test case 'testCancelPayment' is in PaymentInfoControllerTest. So, I have run PaymentInfoControllerTest and the above test result is passed and shown in the below image.



Test Case 7: 'testSuccessPay':

Purpose: The goal is to make sure that the PaymentInfoController provides accurate status code upon creating the payment.

Functionality Under Test: The functionality tested is the execution of the payment and the success response handling by the controller.

Test Steps:

Arrange (Setup the Test Environment): For the endpoint /ppms/payment/{userId}/success, a GET request is set up, but the userId's URI variables are misconfigured as "Uri Variables,".

Act(Execute the Operation): The paymentInfoController is contacted via a GET request without the proper URI variables set up.

Assert(Verify the Outcome): A 400 Bad Request response status is expected, signifying that the request was not properly formed because certain URI variables were either missing or entered incorrectly.

```
/**
 * Method under test:
 * {@link PaymentInfoController#successPay(String, String, long)}
 */
@Test
void testSuccessPay() throws Exception {
    // Arrange
    when(paymentInfoService.executePayment(Mockito.<String>any(), Mockito.<String>any())).thenReturn(new Payment());
    MockHttpServletRequestBuilder requestBuilder = MockMvcRequestBuilders
            .get("/ppms/payment/{userId}/success", "Uri Variables", "Uri Variables")
            .param("PayerID", "foo")
            .param("paymentId", "foo");

    // Act
    ResultActions actualPerformResult = MockMvcBuilders.standaloneSetup(paymentInfoController)
            .build()
            .perform(requestBuilder);

    // Assert
    actualPerformResult.andExpect(MockMvcResultMatchers.status().is(400));
}
```

Inputs: Simulates Values for testing purpose

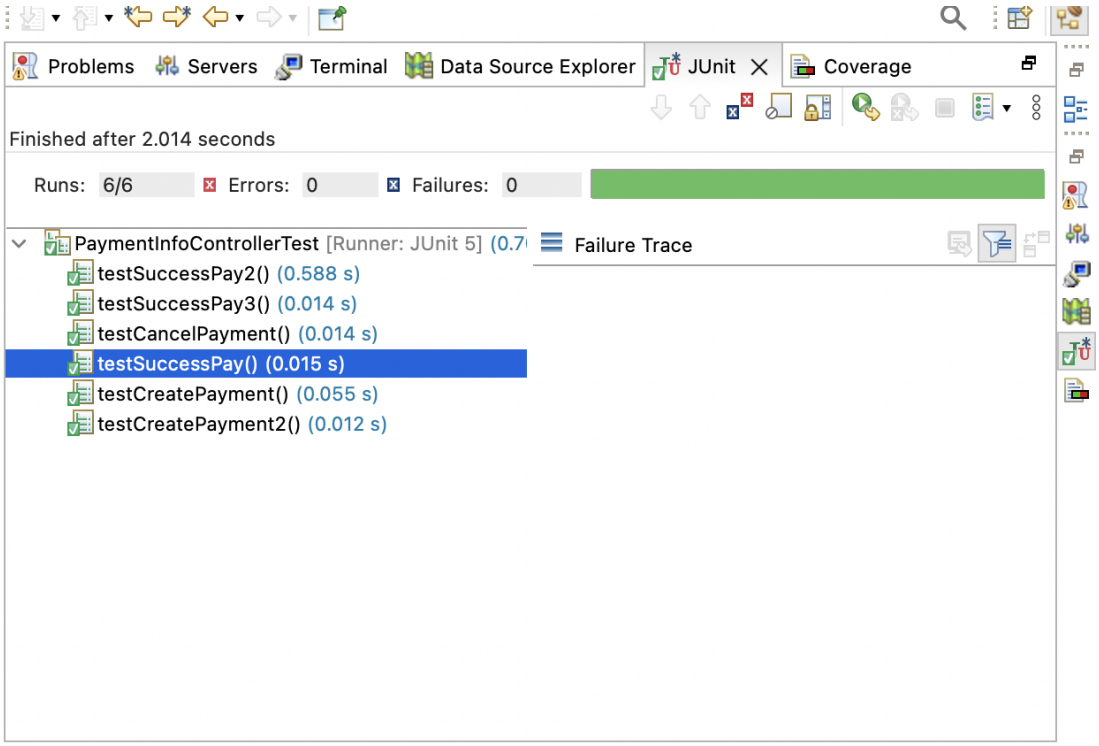
PayerID: "foo"

paymentId: "foo"

Expected Outputs:

HTTP Status: 400 Bad Request

The above test case 'testSuccessPay' is in PaymentInfoControllerTest. So, I have run PaymentInfoControllerTest and the above test result is passed and shown in the below image.



Test Case 8: 'testAddProductToCart':

Purpose: To guarantee that a product is added to a cart via the customerService's addProductToCart function and that the customerController provides the desired confirmation message and content type.

Functionality Under Test: The test verifies the correct response message and content type and determines whether a Cart object is handled correctly when a POST request is performed to add a product to a user's cart.

Test Steps:

Arrange (Setup the Test Environment): With the path variable userId set to 1L, a POST request builder is configured for the endpoint /ppms/customer/{userId}/cart/addproduct.

Act(Execute the Operation): Using MockMvc, the POST request is sent to the customerController, simulating the web context's behavior for testing purposes.

Assert(Verify the Outcome): The response body is verified to contain the string "Add Product to Cart", which is the expected success message. The status should be 200 OK and response content type should be "text/plain;charset=ISO-8859-1"

```java
    /**
     * Method under test: {@link CustomerController#addProductToCart(long, Cart)}
     */
    @Test
    void testAddProductToCart() throws Exception {
        // Arrange
        when(customerService.addProductToCart(Mockito.<Cart>any())).thenReturn("Add Product To Cart");

        Cart cart = new Cart();
        cart.setId(1L);
        cart.setOrderStatus(Cart.OrderStatus.NOT_ORDERED);
        cart.setProductId(1L);
        cart.setProductPrice(10.0d);
        cart.setProductQuantity(1L);
        cart.setUserId(1L);
        String content = (new ObjectMapper()).writeValueAsString(cart);
        MockHttpServletRequestBuilder requestBuilder = MockMvcRequestBuilders
                .post("/ppms/customer/{userId}/cart/addproduct", 1L)
                .contentType(MediaType.APPLICATION_JSON)
                .content(content);

        // Act and Assert
        MockMvcBuilders.standaloneSetup(customerController)
                .build()
                .perform(requestBuilder)
                .andExpect(MockMvcResultMatchers.status().isOk())
                .andExpect(MockMvcResultMatchers.content().contentType("text/plain;charset=ISO-8859-1"))
                .andExpect(MockMvcResultMatchers.content().string("Add Product To Cart"));
    }
```

Inputs: Serialized JSON representation of the Cart object.

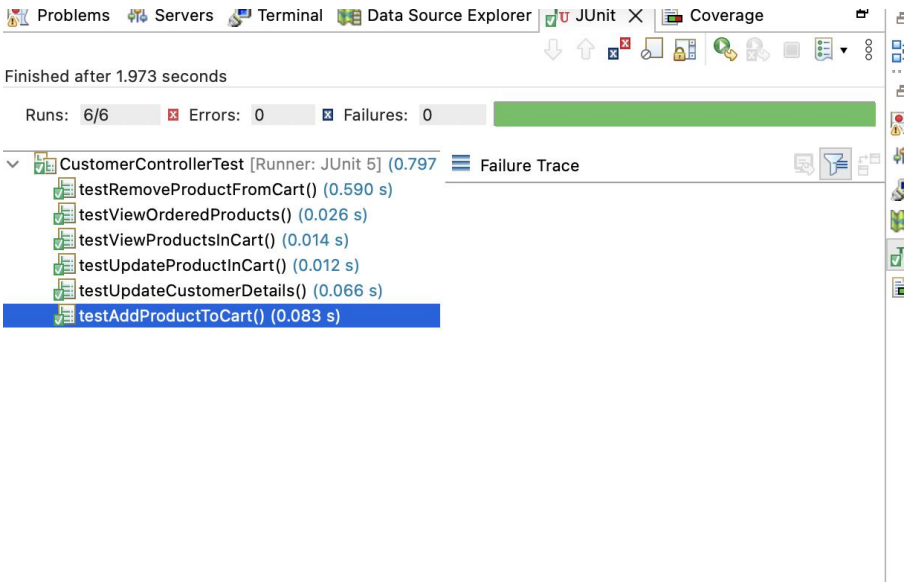
Expected Outputs:

HTTP Status: 200 OK

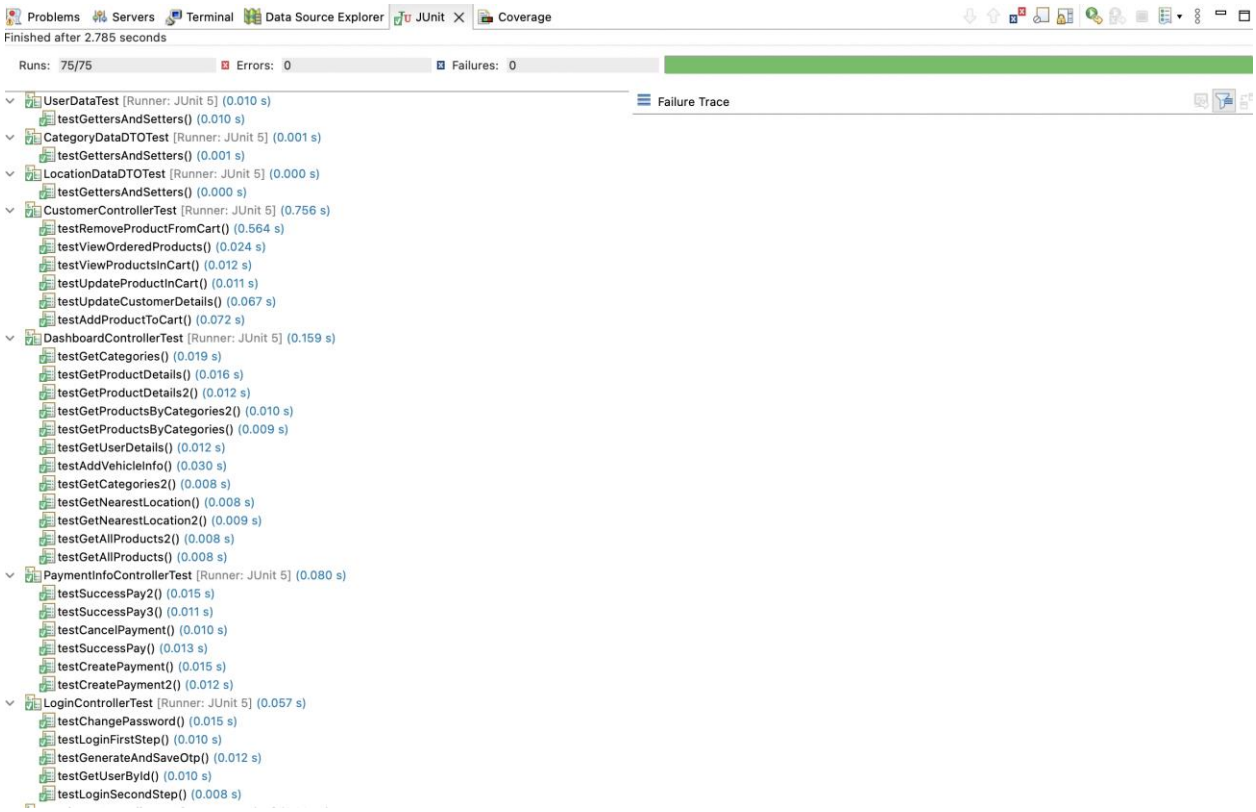
Content Type: "text/plain;charset=ISO-8859-1"

Response Body: "Add Product To Cart"

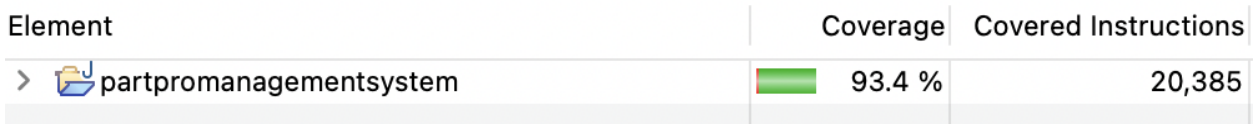
The above test case ‘testAddProductToCart’ is in CustomerControllerTest. So, I have run CustomerControllerTest and the above test result is passed and shown in the below image.



So, there were many more test cases which were written to have code coverage for every line which enhances the performance and stability of the application. The below image shows the results for all the unit tests written.



The below image says that it has covered code coverage of 93.4% which says it has covered almost entire code. So, the application is tested successfully.



D. **<u>Report User Manual</u>**: We have worked on both front end and back end. I will include every step here.

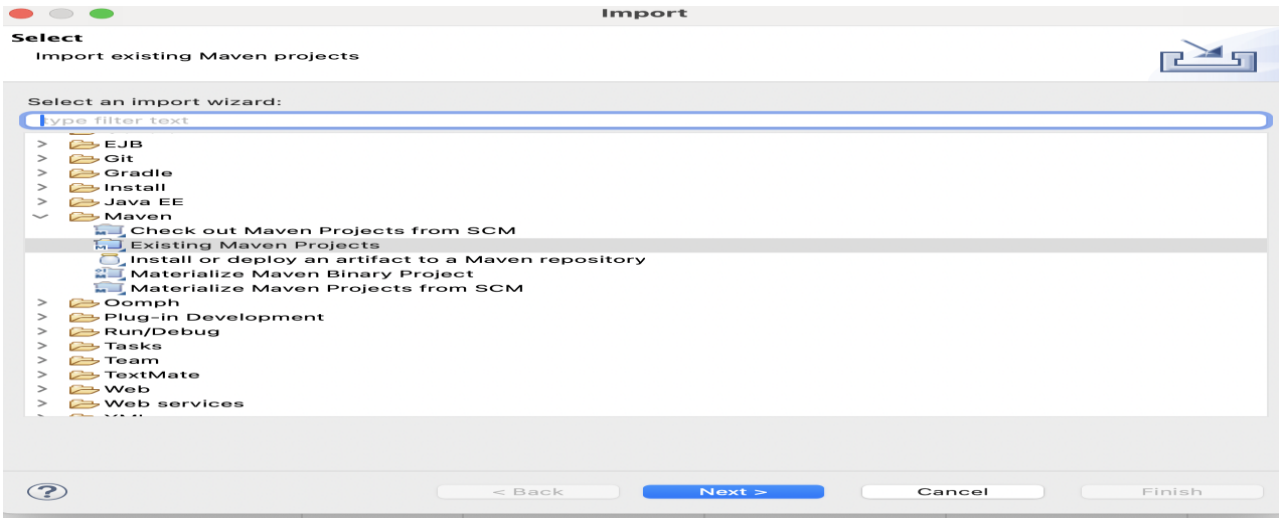
1. Clone the code from the git repository.
   git clone https://github.com/padmarahul/partpro-management-system.git
2. If the git is not recognized in your system, install git from the below link.
   https://git-scm.com/downloads
3. After successful installation of git verify the git version through terminal and the command is git -v.

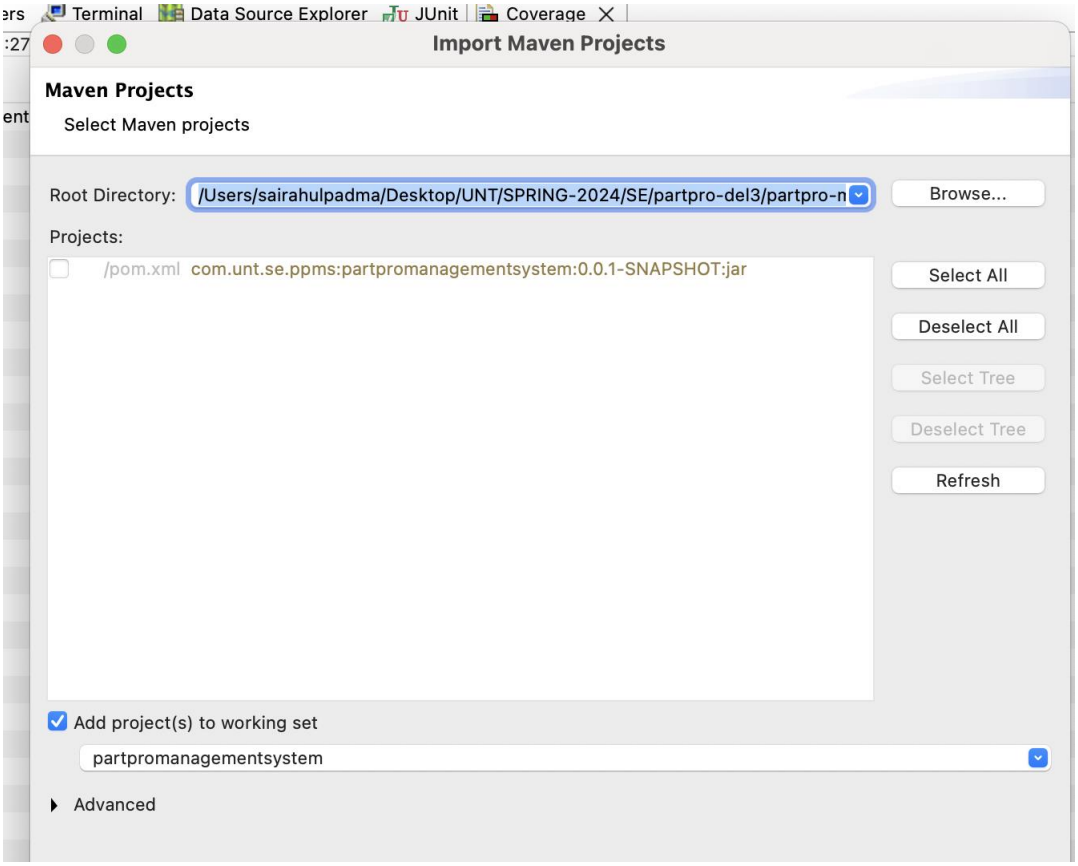
```
sairahulpadma@Sais-MacBook-Air-4 ~ % git -v
git version 2.39.3 (Apple Git-145)
sairahulpadma@Sais-MacBook-Air-4 ~ %
```

4. You need to install all required software for backend. Install eclipse IDE for back-end code development. Install Postman for API Testing. Install MySQL Workbench for data storage. You can download all the mentioned software's from the below links.
   Eclipse Installation: https://www.eclipse.org/downloads/packages/
   MySQL Workbench Installation: https://dev.mysql.com/downloads/workbench/
   Postman Installation: https://www.postman.com/downloads/
5. You need to install JDK for getting java environment and development kits. Install jdk from below link.
   https://www.oracle.com/java/technologies/javase/jdk21-archive-downloads.html
6. After successful installation of jdk verify the java version through terminal. And the commands are java –version and javac –version

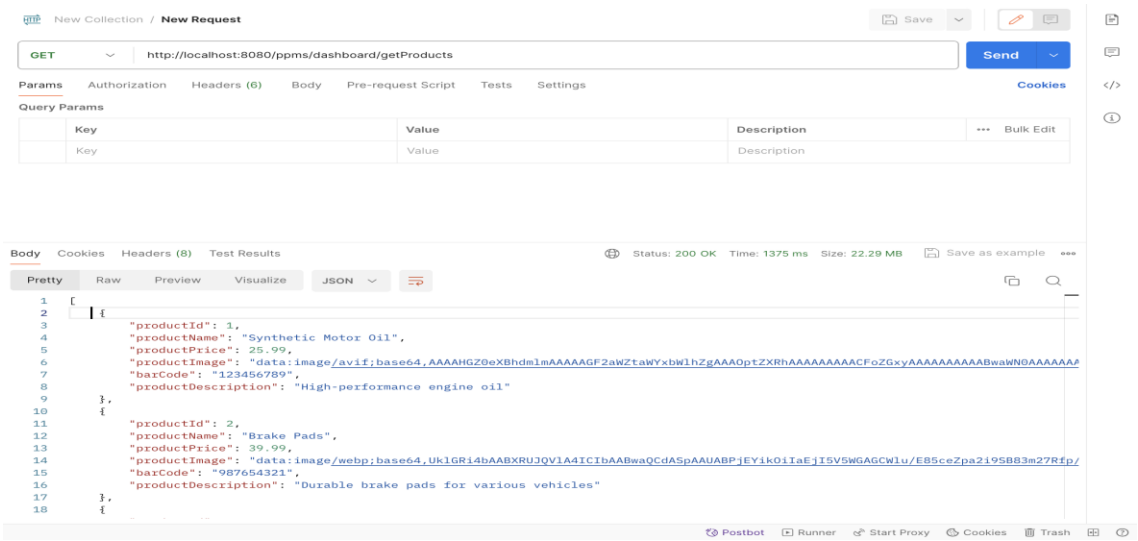
```
sairahulpadma@Sais-MacBook-Air-4 ~ % java --version
java 21.0.2 2024-01-16 LTS
Java(TM) SE Runtime Environment (build 21.0.2+13-LTS-58)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.2+13-LTS-58, mixed mode, sharing)
sairahulpadma@Sais-MacBook-Air-4 ~ % javac --version
javac 21.0.2
sairahulpadma@Sais-MacBook-Air-4 ~ %
```

7. Go to the directory till src/backend and import the back-end repository into eclipse.
   Click on File→ Import → Existing Maven Projects.
   Add the root directory till src/backend and click Finsh. The back end code is imported successfully.





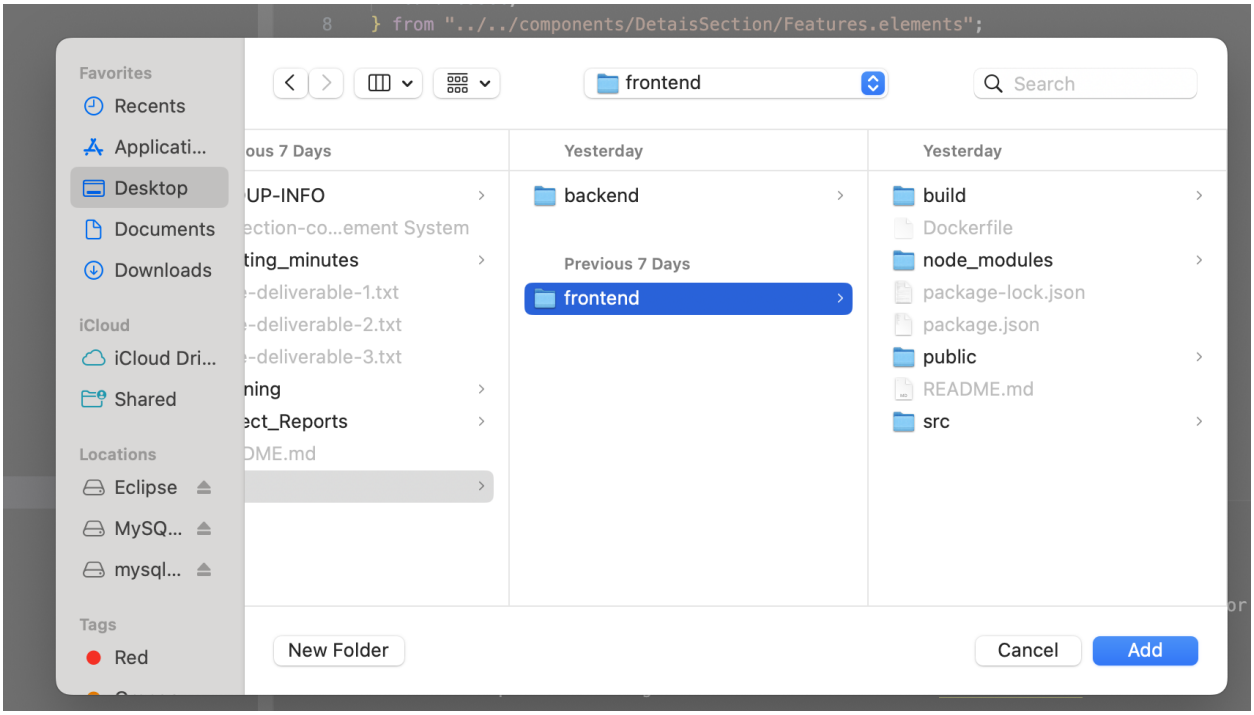
8. Once the application is up and running test the APIs through postman. We can also test all the API through swagger. The swagger is used to store all the API collection and this configuration is written in the back-end code. The swagger link would be running once the application is built successfully and the link is pasted below.
   http://localhost:8080/swagger-ui/index.html
9. We can also test API through postman.



10. Now install the front-end software's for working on front end code. Install the Node JS for running front end on web server and it installs all required dependencies. Install Visual studio code for front end code development.
    Node JS Installation: https://nodejs.org/en/about/previous-releases
    Install Node JS of 14.x.x for compatibility.
    Visual Studio Code Installation: https://code.visualstudio.com
11. Verify the node version after installation through terminal and the commands are node -v and npm -v

```
sairahulpadma@Sais-MacBook-Air-4 ~ % npm -v
6.14.13
sairahulpadma@Sais-MacBook-Air-4 ~ %
```

12. Go to the directory till src/frontend and import the front-end repository into vs code.
    Click on File→ Add Folder to Workspace
    Add the root directory till src/frontend and click Add. The front-end code is imported successfully.

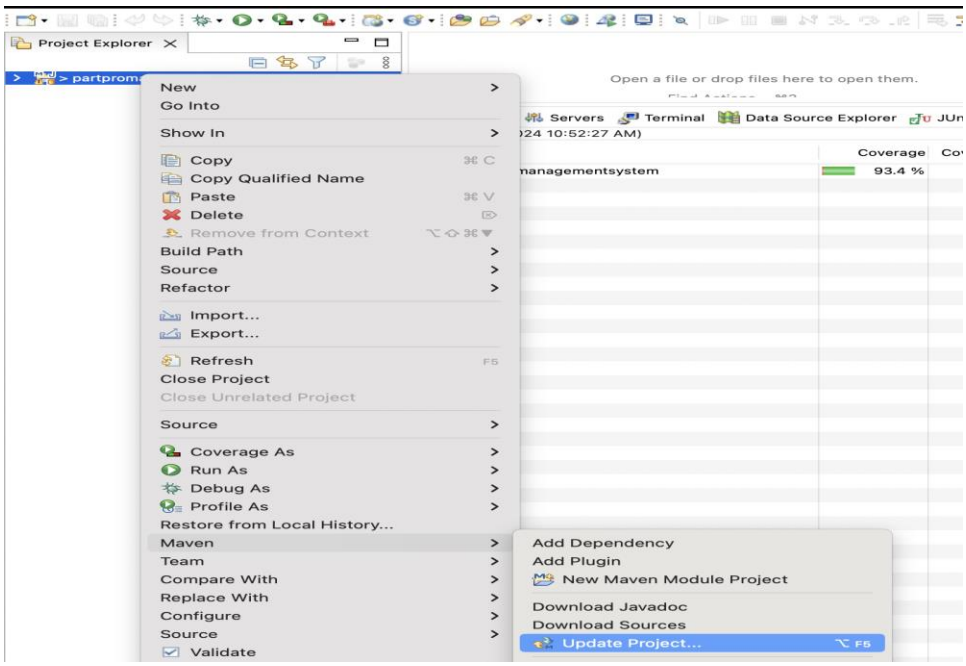


13. After the application is up and running use the application through web link. The link is pasted below.
    http://localhost:3000/

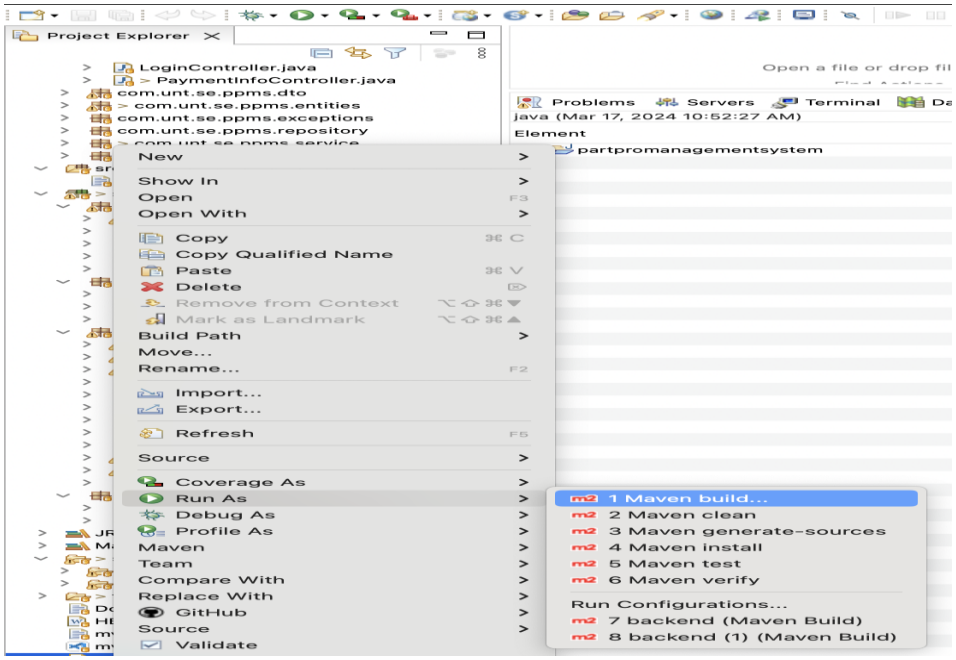
E. **Report Compilation Instructions**:

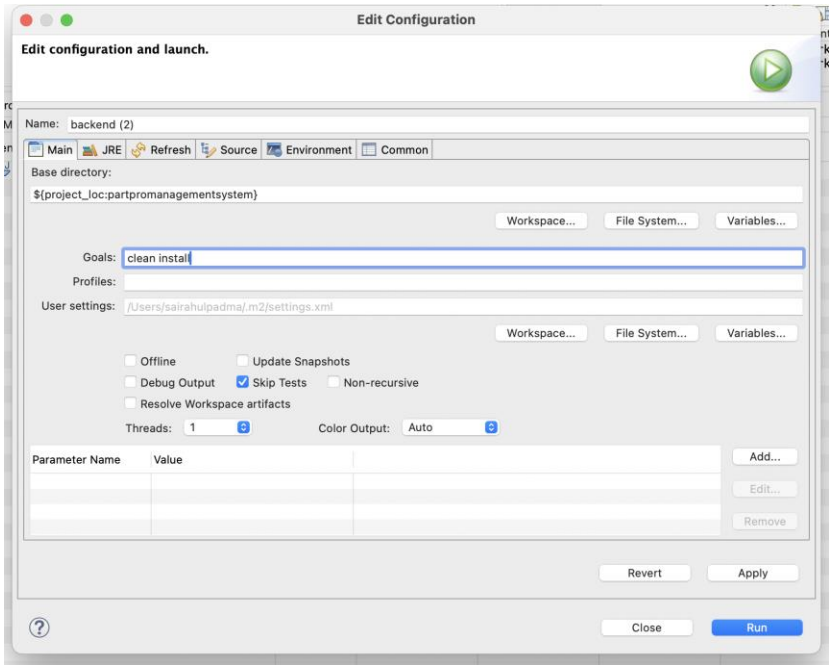
For Backend:

1. Once the code is imported successfully, update the maven project which would refresh the back-end code.



2. Compile and build the project by using maven build option and click on Run.

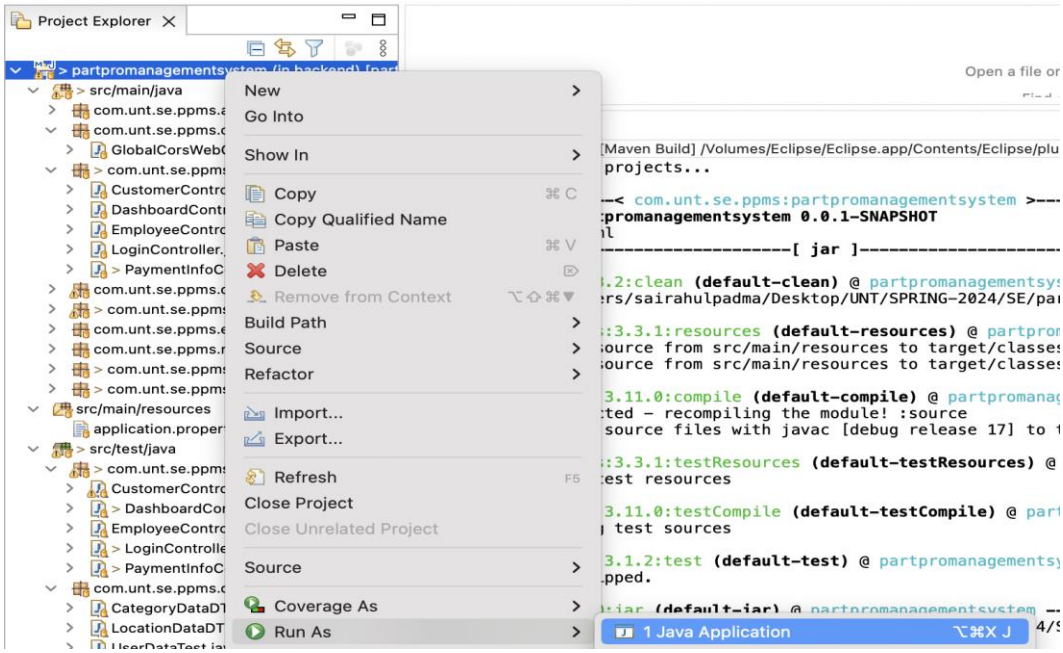




3. You would get the below output in the console.

```
[INFO] Scanning for projects...
[INFO]
[INFO] -------------< com.unt.se.ppms:partpromanagementsystem >---------------
[INFO] Building partpromanagementsystem 0.0.1-SNAPSHOT
[INFO]   from pom.xml
[INFO] --------------------------------[ jar ]---------------------------------
[INFO]
[INFO] --- clean:3.3.2:clean (default-clean) @ partpromanagementsystem ---
[INFO] Deleting /Users/sairahulpadma/Desktop/UNT/SPRING-2024/SE/partpro-del3/partpro-management-system/src/backend/target
[INFO]
[INFO] --- resources:3.3.1:resources (default-resources) @ partpromanagementsystem ---
[INFO] Copying 1 resource from src/main/resources to target/classes
[INFO] Copying 0 resource from src/main/resources to target/classes
[INFO]
[INFO] --- compiler:3.11.0:compile (default-compile) @ partpromanagementsystem ---
[INFO] Changes detected - recompiling the module! :source
[INFO] Compiling 53 source files with javac [debug release 17] to target/classes
[INFO]
[INFO] --- resources:3.3.1:testResources (default-testResources) @ partpromanagementsystem ---
[INFO] Not copying test resources
[INFO]
[INFO] --- compiler:3.11.0:testCompile (default-testCompile) @ partpromanagementsystem ---
[INFO] Not compiling test sources
[INFO]
[INFO] --- surefire:3.1.2:test (default-test) @ partpromanagementsystem ---
[INFO] Tests are skipped.
[INFO]
[INFO] --- jar:3.3.0:jar (default-jar) @ partpromanagementsystem ---
[INFO] Building jar: /Users/sairahulpadma/Desktop/UNT/SPRING-2024/SE/partpro-del3/partpro-management-system/src/backend/target/partpromanagementsystem-0.0.1-SNAPS
[INFO]
[INFO] --- spring-boot:3.2.2:repackage (repackage) @ partpromanagementsystem ---
[INFO] Replacing main artifact /Users/sairahulpadma/Desktop/UNT/SPRING-2024/SE/partpro-del3/partpro-management-system/src/backend/target/partpromanagementsystem-0
[INFO] The original artifact has been renamed to /Users/sairahulpadma/Desktop/UNT/SPRING-2024/SE/partpro-del3/partpro-management-system/src/backend/target/partpro
[INFO]
[INFO] --- install:3.1.1:install (default-install) @ partpromanagementsystem ---
[INFO] Installing /Users/sairahulpadma/Desktop/UNT/SPRING-2024/SE/partpro-del3/partpro-management-system/src/backend/pom.xml to /Users/sairahulpadma/.m2/repositor
[INFO] Installing /Users/sairahulpadma/Desktop/UNT/SPRING-2024/SE/partpro-del3/partpro-management-system/src/backend/target/partpromanagementsystem-0.0.1-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  3.483 s
[INFO] Finished at: 2024-03-17T11:52:36-05:00
[INFO] ------------------------------------------------------------------------
```

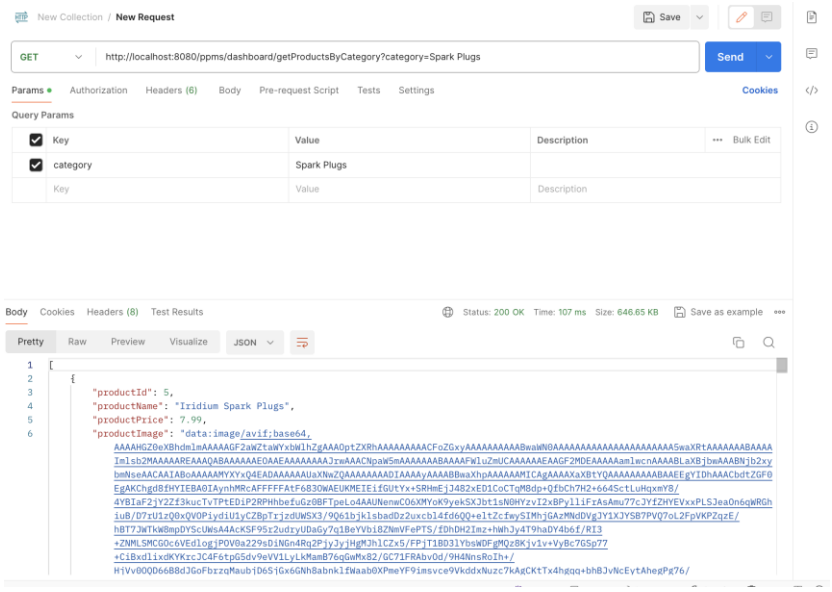
4. Run the java application once the application is compiled successfully.



5. You would get the below output in the console and the application would be running on embedded tomcat server on 8080 port.

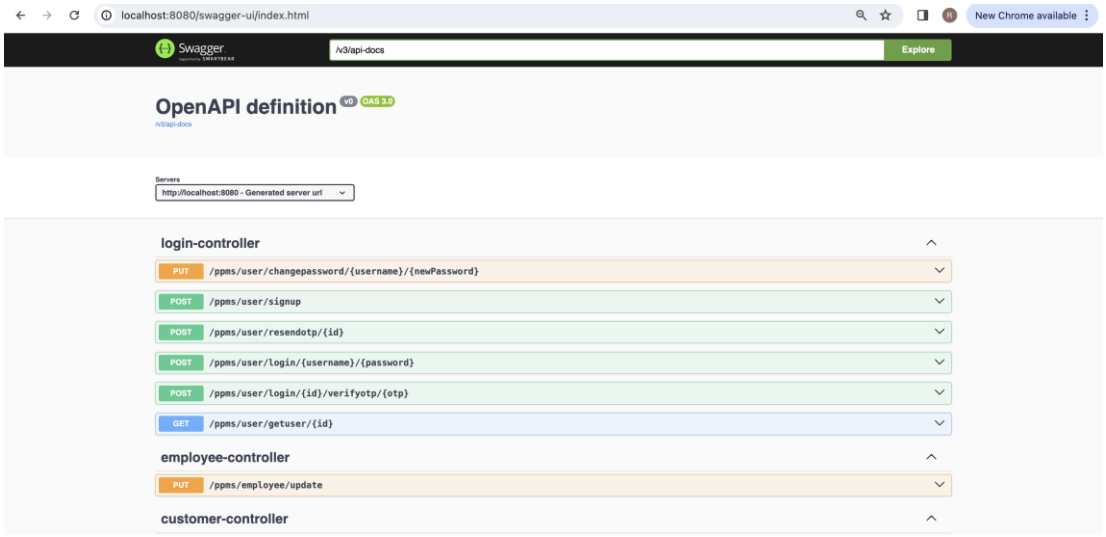
```
2024-03-17T11:56:01.863-05:00  INFO 29987 --- [  restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-03-17T11:56:02.073-05:00  INFO 29987 --- [  restartedMain] o.s.d.j.r.query.QueryEnhancerFactory     : Hibernate is in classpath; If applicable, HQL parser will be used.
2024-03-17T11:56:02.755-05:00  WARN 29987 --- [  restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries
2024-03-17T11:56:03.260-05:00  INFO 29987 --- [  restartedMain] o.s.b.d.a.OptionalLiveReloadServer       : LiveReload server is running on port 35729
2024-03-17T11:56:03.266-05:00  INFO 29987 --- [  restartedMain] o.s.b.a.e.web.EndpointLinksResolver      : Exposing 1 endpoint(s) beneath base path '/actuator'
2024-03-17T11:56:03.318-05:00  INFO 29987 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port 8080 (http) with context path ''
2024-03-17T11:56:03.332-05:00  INFO 29987 --- [  restartedMain] s.p.a.PartpromanagementsystemApplication : Started PartpromanagementsystemApplication in 5.003 seconds (process runnin
```

6. Test the API through postman once the back-end application is up and running. I have pasted sample API which was tested in postman.



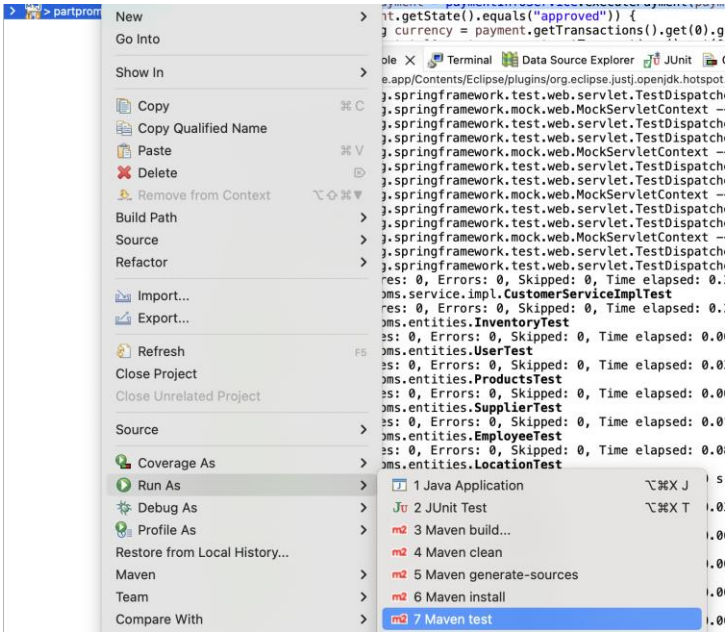
7. If you need all the API collection, Run the swagger link in your browser. We need to run the application before opening Swagger URL.

Swagger URL Link: http://localhost:8080/swagger-ui/index.html

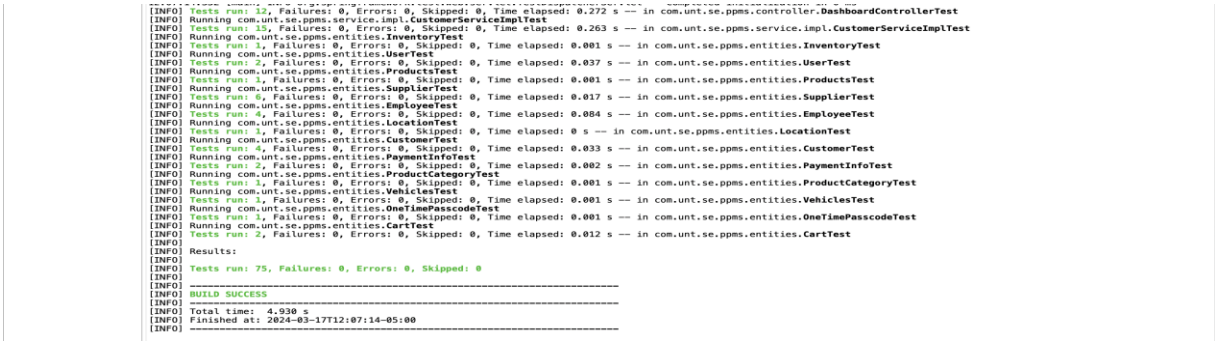


For Backend Test cases:

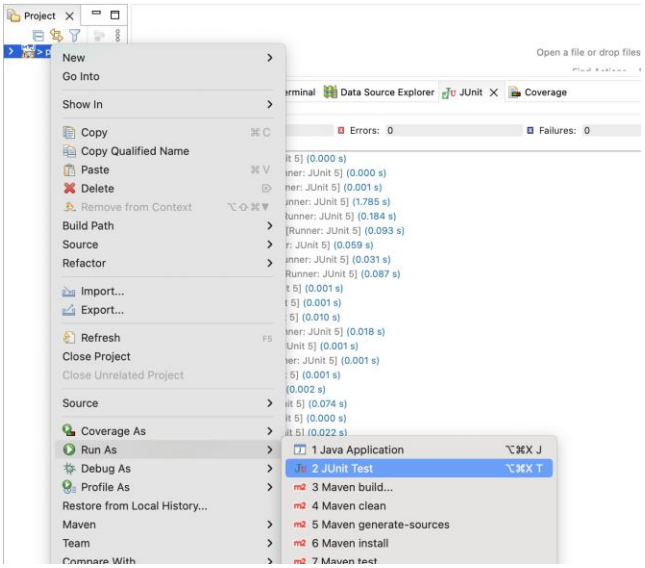
1. Compile the junit tests which were in src/test/java. Compile the test cases as shown below.



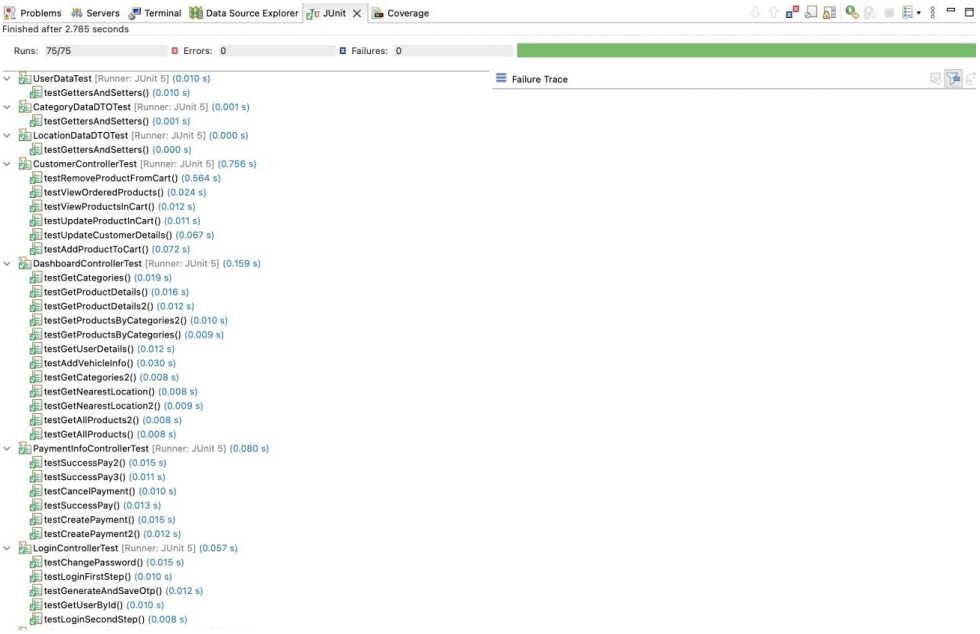
You would get the below output from console.



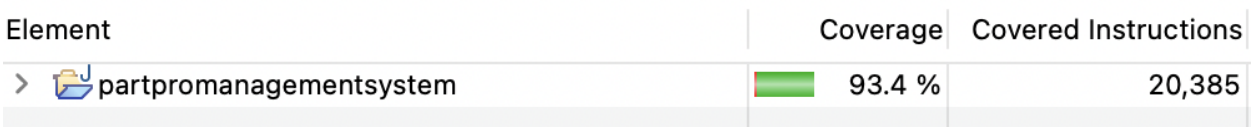
2. Run the junit tests which were in src/test/java. Run the test cases as shown below.



You would get the below output from the console.

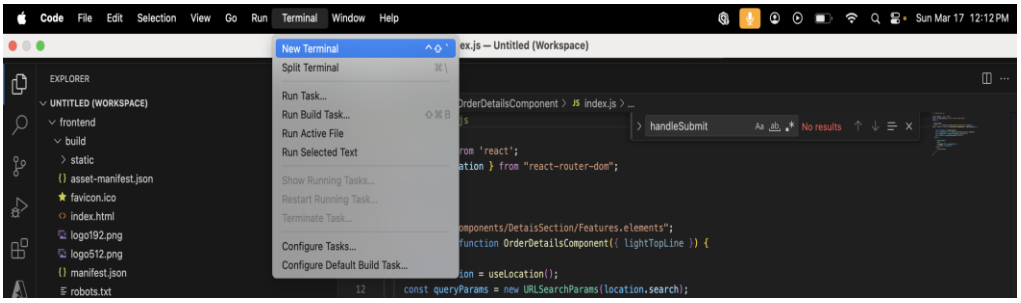


3. Coverage of code can be checked by clicking on Coverage As option on project name. You would get the below response from the coverage screen.



For Front end:

1. Open the terminal in vscode as shown below.



2. Install all the required dependencies by using npm install command and you would get the output as shown below.

```
audited 1874 packages in 7.934s

145 packages are looking for funding
  run `npm fund` for details
```

3. Build the front end application by using command npm run build and you would get the output as shown below.

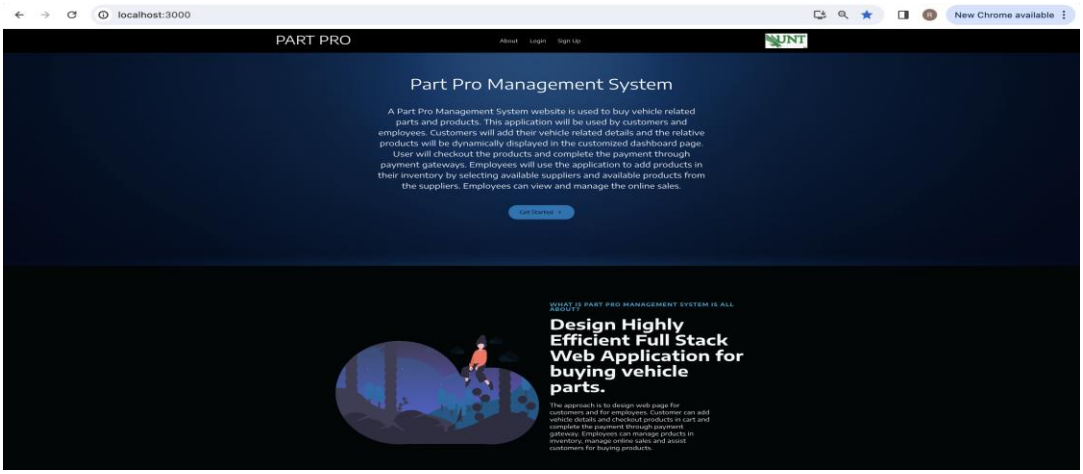
```
The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

  https://cra.link/deployment
```

4. Start the front-end server by using this command npm start and when you access the application with the web URL ‘http://localhost:3000’ you would get the output as shown below. We can start using the application once the front-end application is up and running.



F. Report Peer Review Feedback:

Summary:

PartPro team have introduced us and we have started explaining regarding our project introduction. They have asked some questions on our project design, description and regarding the software technology stack used. They have asked us questions like How your application is getting deployed, What is the database you are currently using, How you are going to test your application for back end and front end, What kind of roles are assigned to an employee entity in your design, What kind of validations are you performing on customer personal data, How you are going to do the payment linkage with external payments, What kind of permissions and levels are applicable to each employee type, How user gets nearest outlet or part pro store, Are you going to track all the orders and collect feedback from the customers. We have answered those questions in brief.

Suggestion Given:

We have got couple of suggestions to be included as part of future deliverables like we can add Order Management, order tracking, rating, and feedback and live chatbot integration, coupon, and discounts for the existing product. We have rejected live chatbot integration and accepted the rest of functional requirements. We have discussed the design approach for the accepted functional requirements. They have concentrated on a couple of non-functional requirements such as code readability and code maintainability which we have already planned to implement during each deliverable.

Actions Taken based on feedback:

In the personal team meeting, we discussed the suggestions they have given and tried to discuss the baseline design and low-level design regarding how to implement those suggestion in our project. They have asked us to add Order Tracking of the orders once user completes the payment. They have asked us to add the Feedback module, by taking the customer rating once user lands on order details page. They have asked us to add discount and coupons feature by adding loyalty points as a custom field for the customer. We have discussed our timelines and thought of implementing these suggestions once the front end , back end, testing and deployment has done. We have discussed couple of new entities related to feedback and orders and their mapping between the existing entities. We have done brainstorming on what attributes these entities should contain. Hence, we have concluded the call once the discussions regarding the suggestions were done. We have accepted to implement Order Management, order tracking, rating, and feedback, coupon, and discounts and rejected chatbot integration. For code maintainability and code readability, we have planned to add code comments and code modularization.

G: **Report Reflection:**

Accomplishments:

1. **Feature Implementation**: Successfully implemented key features like Login, Signup, Multi factor authentication using OTP, forgot password, retrieving all products, retrieving all categories, retrieving nearest store location details, add and removal of items into cart. Completing and cancelling payment.
2. **API Development**: All the API's which are developed are REST API'S and exposed in back-end spring boot controller layer and exception handling is done for all the API's.
3. **Junit Testing:** Unit tests are written for all the packages and code coverage is greater than 90%.
4. **API Testing**: API'S are tested using Postman which helped us for integrating APIs in front end.
5. **UI Screens Design**: Designed pages for all the core features which we have planned to implement for development phase 1. Reusable components are also created for code reusability.

Positive Aspects which went well:

1. Design of UML diagrams like class diagrams, use case diagrams for normal and error case and sequence diagrams helped us to have a better walk functional walk through of the application.
2. The adoption of agile methodology is very effective and facilitated positive outcome and completed the project within the deadline.
3. Code reviews and feedbacks were taken on the developed code and raised the new revisions which helped us to build the application better.
4. Team collaboration and communication are the key aspect that went well. We have divided the core features among ourselves in front end and back end and pushed the code to main branch.
5. The testing has also gone well by performing API testing and Junit testing.

Areas of Improvement:

1. Code comments and documentation should be done better for the developed code.
2. Integration and end to end UI testing should be done instead of functional tests which we have done in this phase.
3. Requirements gathering for the next features development should be done more effectively before starting development to avoid multiple revisions of the code.
4. In future development phases, we need to concentrate on finding all the blockers before starting development which would enhance time management.

H. Report Member Contribution Table:

Member name	Contribution description	Overall Contribution (%)	Note (if applicable)
Sai Rahul Padma	I have worked on initial back-end code setup, initial front end code setup, design of Use case, sequence , class diagrams, and database setup and written unit tests for developed code.	14.5	
Nikhita Muvva	I have worked on front end dashboard related pages design, routes to dashboard pages, service API integrations and required component and CSS designs.	12.5	
Sai Samyuktha Paspuleti	I have worked on back-end customer and payment related API development, entities creation, exception handling and service implementation.	12	
Rishika Yalamanchili	I have worked on front end cart and payment related pages design, routes to cart and payment pages, service API integrations and required component and CSS designs.	12	
Pavani Venigalla	I have worked on back-end login, OTP and signup related API development, entities creation, exception handling and service implementation.	12.5	

Himabindu Chunduri	I have worked on front end customer related pages design, routes to customer pages, service API integrations and required component and CSS designs.	12	
Sneha Reddy Gangannagari	I have worked on back-end dashboard related API development, entities creation, exception handling and service implementation.	12.5	
Jhasha Sri Ede	I have worked on front end login related pages design, routes to login pages, service API integrations and required component and CSS designs.	12	