Deliverable 5:

A. Report Requirements:

List of Functional and Non-Functional Requirements for Development Phase 3:

1. Back-end REST API Development Functional Requirements:

1.1 Adding Products in Inventory Based on Inventory Levels: Store Owner and Store Manager can access the inventories in stores. The API endpoint is exposed in Employee Controller. The back end database call would trigger from service layer to Inventory repository layer. The stock status would become out of stock once the product quantity is zero. The out of stock products are listed and sent as back end response.

1.2 View Online Sales: Store owner can access to online sales performed in a store. Once customer completes the payment, online sales are stored in the database. The API endpoint is exposed in Payment Controller. The back end database call would trigger from service layer to Online Sales repository layer. Each online sales tuple contains the product ID, sales ID, customer ID, employee ID and total amount. Another API is developed to view all the online sales performed in a store. It fetches the data from online sales entity.

1.3 Add and Update Feedback: Customer can add and update feedback for the products which are available in a store. The API endpoint is exposed in Customer controller layer. The back end database service call would trigger from Customer controller layer. If the feedback rating already exists for a product, the average of the rating is done based on rating counts. Feedback entity contains feedback ID, product ID, rating, and rating count. The tuple gets persisted into DB. Database service call would trigger from service layer to Feedback repository layer. If the feedback rating does not exist for the product, the new feedback tuple gets persisted into DB. Another API is developed to view the feedback of the products. It fetches the data from feedback entity.

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

2.1 Adding Products in Inventory Based on Inventory Levels: Manage Inventory Levels Component is created in pages folder and specific route is created to navigate to web page. When employee clicks on Update Inventory By Stock Status button on navbar, user would land on the inventory levels screen. Employee can only view products which are out of stock. The employee can click on any of the listed products. Once employee clicks on it, the popup would be appeared where employee needs to enter quantity. Once the employee clicks on submit button, the inventory gets updated with stock status as true and with the quantity what user has entered. It triggers the back end API call which is integrated in service folder.

2.2 View Online Sales: Manage Online Sales Component is created in pages folder and specific route is created to navigate to web page. When store owner clicks on manage online sales button on navbar, user would land on manage online sales screen. Store owner can view all the online sales performed in a store in tabular view. It contains the sale ID, customer Name, employee Name, product Name and the total amount. The data is dynamically loaded and it fetched from back end API which is triggered in service folder.

2.3 Add and Update Feedback: Manage Feedback component is created in pages folder and specific route is created to navigate to web page. When customer clicks on Add Feedback from the navbar, user would land on feedback screen. User can provide feedback for products available in store. Once user clicks on any of the product, the popup would appear where user provides the rating. Once user clicks on submit button, the feedback gets updated in back end. User can also view feedback by clicking on view feedback button available at each product.

3. Other Non-Functional Requirements:

3.1 Code reviews and merge requests for phase 3 developed code: The code has been written for new use cases which are developed for phase 3. Code has been written for both front end and back end. Code reviews are done for all the merge requests. Code comments are done and raised new revisions and merged the code to main branch.

3.2 Code deployment for phase 3 developed code: The phase 3 developed code has been deployed to Azure cloud VM. The latest code has been taken from main branch and the docker images were newly created for both front end and back end. Docker containers are stopped initially before docker images are created. Once the updated docker images are created, we need to start the docker containers. The docker containers are up and running successfully and verified the same through docker logs.

3.3 Optimization:

Performance Improvement: To improve data retrieval and decrease server response times, the application makes use of Spring Boot caching methods and MySQL indexing. These steps are essential to ensuring that user requests are handled effectively.

Frontend Optimization: React JS components are optimized to increase the user interface's responsiveness and interaction quality. To improve user experience, this involves reducing pointless re-renders and using virtualized lists to manage big datasets.

3.4 Security:

Secure Data Transmission: All data transferred between the client and server is encrypted thanks to the application's enforcement of HTTPS usage. This is necessary to ensure that user data is private and secret throughout transmission by guarding against man-in-the-middle attacks and eavesdropping. Load balancers are implemented for secured data transmission.

SQL Injection Prevention: Prepared statements and strict input validation features built into Spring Boot are used by the application to counter SQL injection threats. These

procedures are essential for avoiding harmful data from impacting the database's availability or integrity.

3.5 Performance:

Server-Side Caching: Spring Boot incorporates caching methods to temporarily store data that is accessed often. This method reduces the need for several database queries, which speeds up server response times and improves the application's overall responsiveness.

Effective Frontend Rendering: React JS is used by the application to optimize component rendering. By ensuring that the frontend runs smoothly and effectively, methods like reducing pointless re-renders and utilizing lazy loading are used to improve interaction behavior and page loading speeds.

3.6 Scalability:

Database Scalability: The MySQL database configuration makes use of techniques like replication and partitioning to disperse demand and improve data management effectiveness, guaranteeing the database scales well with the application.

Stateless Design: The backend, created with Spring Boot, follows a stateless design, which makes scaling easier because individual server instances may react to requests without requiring knowledge about prior interactions or user sessions.

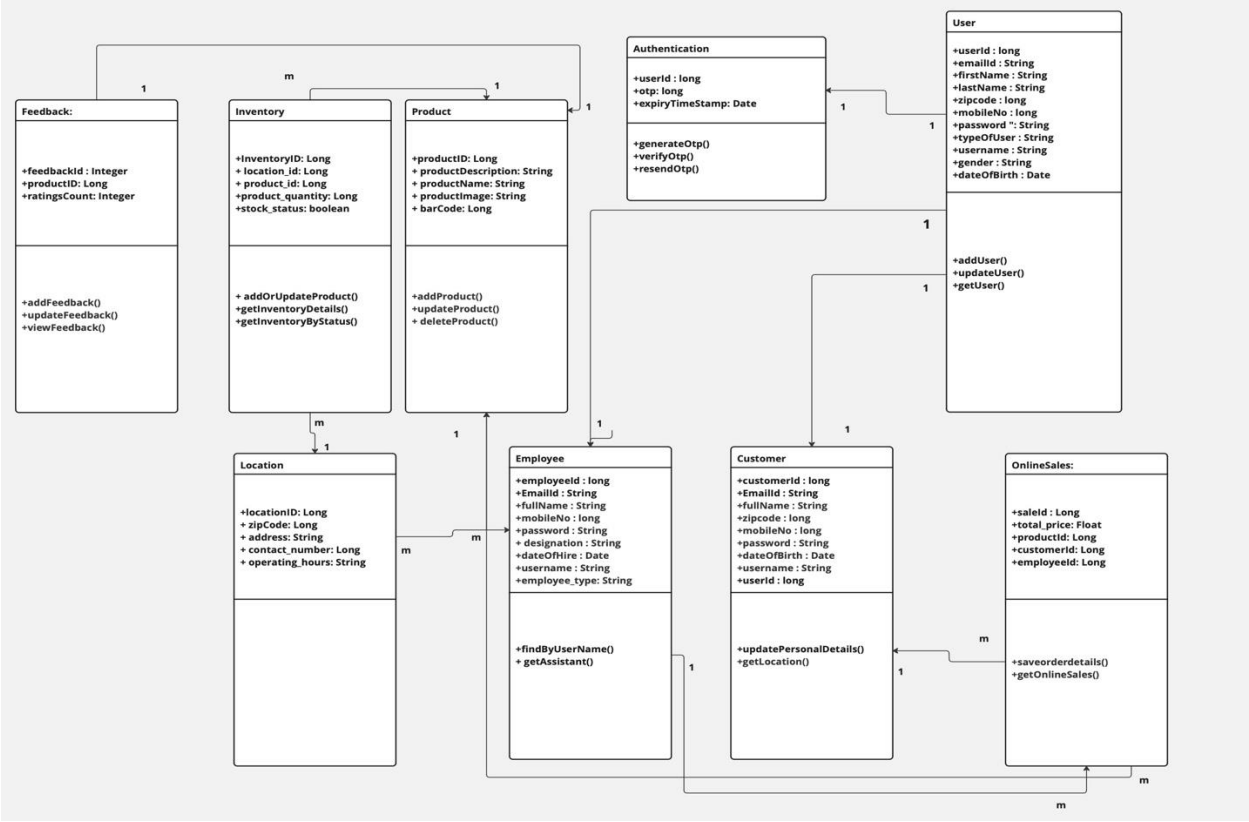
3.7. Usability:

User Interface Design: To accommodate a diverse user base, including individuals with disabilities, the program features a clear and simple React JS interface that emphasizes accessibility and ease of use.

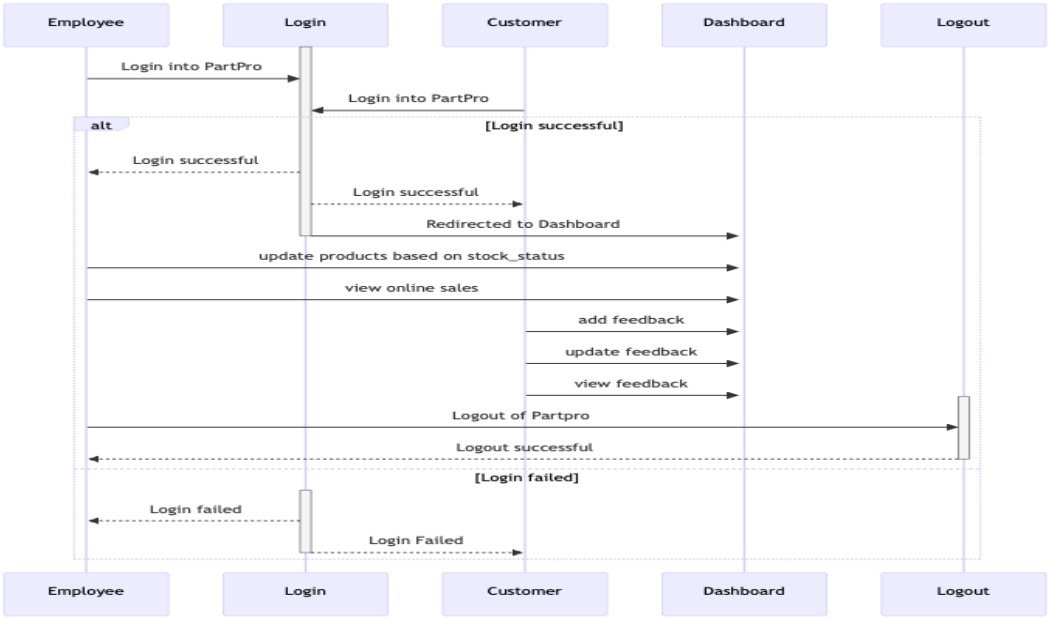
Feedback Mechanisms: Interactive feedback techniques are incorporated into the application to provide users with real-time guidance. Indicators of the continuing procedure, confirmations, and error messages all aid in reducing user error and improving user experience in general.

B. Report UML:

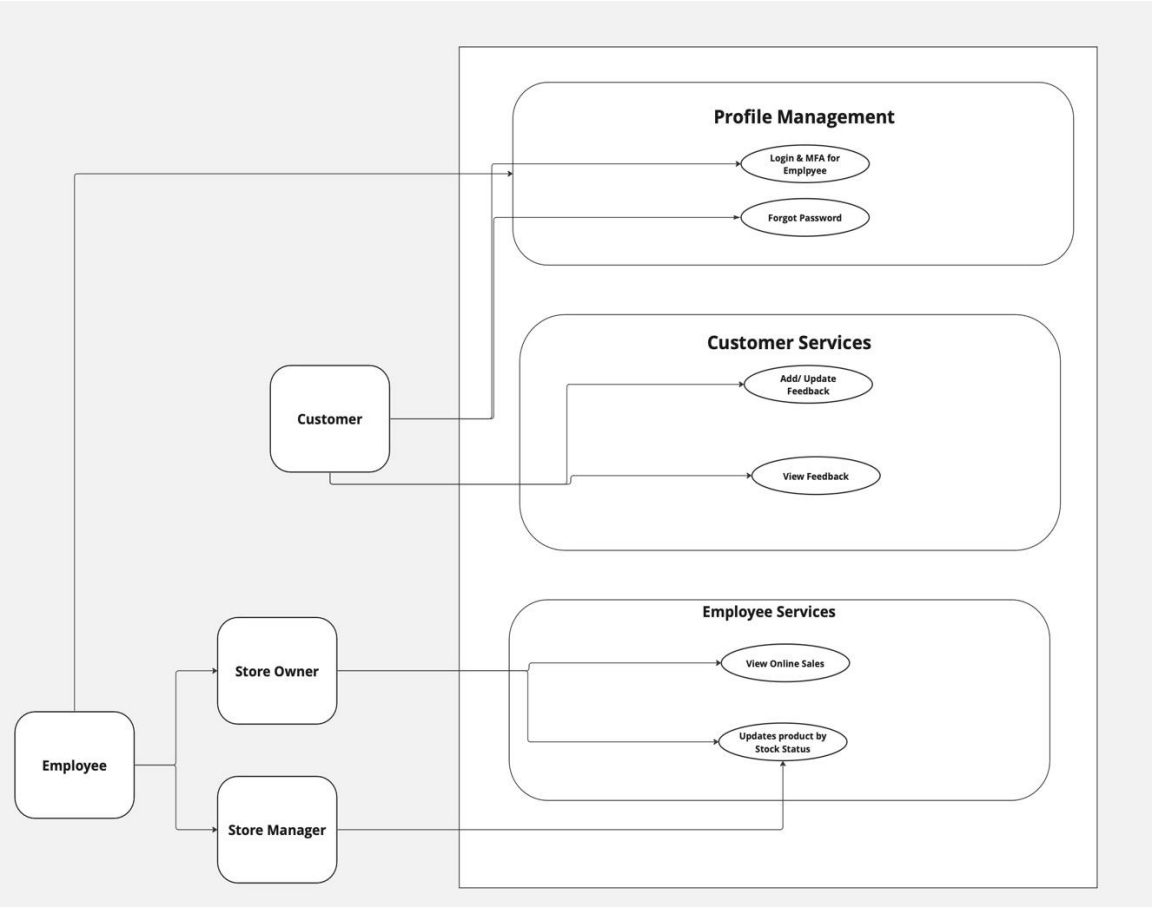
Class Diagram: This development phase includes all other left functionalities of employee and customer. The classes used are Employee, Customer, Feedback, Online Sales, Inventory and Product. One feedback is linked to single product. The mapping between feedback and product is One to One. Online sales are linked to product, employee, and customer. One customer can be part of multiple online sales. One employee can be part of multiple online sales. One product can be part of multiple online sales. The mapping between employee and online sales is One to Many. The mapping between customer and online sales is One to Many. The mapping between products and online sales is One to Many. Multiple products can be part of an inventory. Mapping between inventory and products is Many to One.

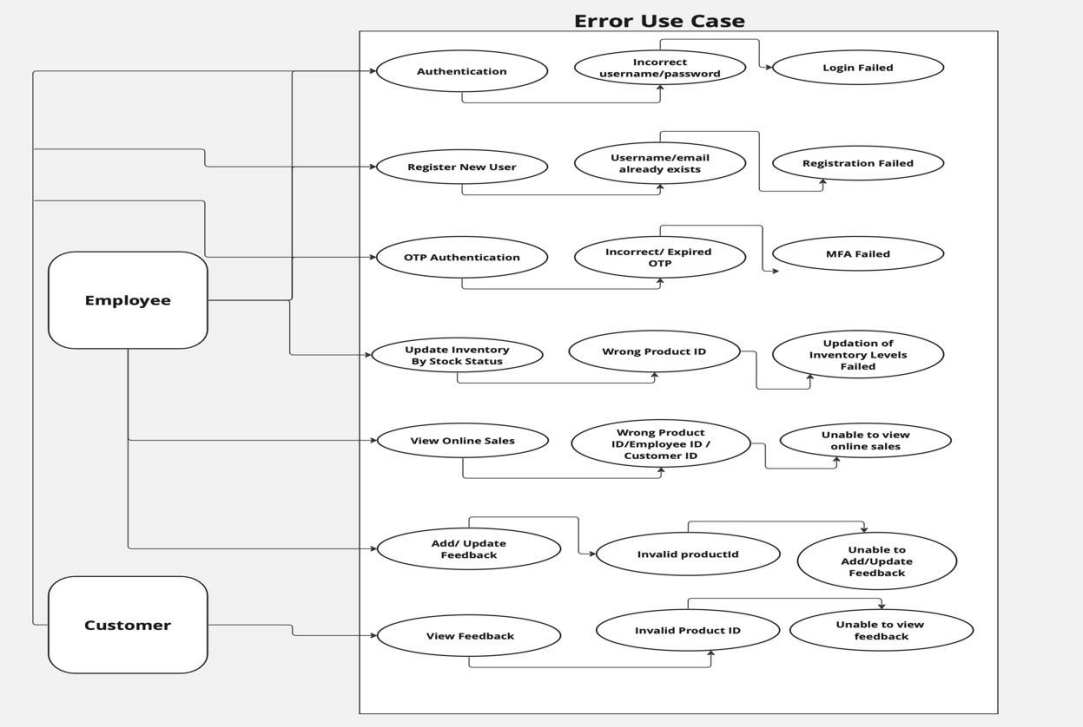


Sequence Diagram: Once customer logs into the application, customer would land on dashboard page. Once customer clicks on add feedback, feedback screen would be appeared. Customer can also view feedback by clicking on view feedback button. Once employee logs into the application, employee would land on dashboard page. Once employee clicks on manage online sales, online sales screen would be appeared. Once employee clicks on update inventory levels, user would land on screen where the user can view products which are out of stock in a store.



Use Case Diagram: The employees would be employees and customers. The employee would perform various actions like update inventory by stock status, manage online sales. Store owner can perform both actions. Store Manager can only perform update inventory by stock status. The customer would perform various actions like add feedback, update feedback and view feedback. The negative error use case diagram includes update inventory levels by stock status. If the product ID is wrong, Updation of Inventory levels failed. It also includes add/update, view feedback. If the product ID is wrong, user cannot view, update, and add feedback. If the employee ID, customer ID and product ID is wrong, user cannot view online sales.





C. Report Test Cases:

I have written test cases and tested features which were developed as part of development phase 3. The test cases are written for better code quality and code coverage. The test cases are written for all the controller endpoints which were exposed in Controller layer. System tests are also integrated for testing all the multiple components which are developed as part of all development phases.

Test Case 1: testAddOrUpdateFeedback

Purpose: This test case aims to verify the functionality of the addOrUpdateFeedback method in managing the addition or updating of feedback for products.

Functionality Under Test: The test evaluates whether the method correctly adds new feedback or updates existing feedback for a product based on the provided productId, rating, and associated userId.

Test Steps:

1. Mock the customerService.addOrUpdateFeedback method to return a predefined success message.
2. Perform a POST request to the /ppms/customer/{userId}/addOrUpdateFeedback endpoint, passing userId, productId, and rating as parameters.
3. Assert that the HTTP response status is OK (200).
4. Assert that the response content matches the expected success message "Feedback updated successfully".

Expected Inputs:

1. userId: 1L (Long)
2. productId: 1L (Long)
3. rating: 4.5f (float)

Expected Output:
1. A success message, "Feedback updated successfully", confirming the correct execution of the feedback addition or update process.

Test Case 2: testViewOnlineSales

Purpose: This test case aims to verify that the viewOnlineSales method accurately retrieves and displays a list of online sales. It ensures that the method effectively handles the retrieval process and correctly formats the returned data as JSON.

Functionality Under Test: The test checks if the viewOnlineSales method can fetch an array of online sales information, and if this data is properly returned in the response body as JSON. It also verifies that the HTTP status code is set correctly.

Test Steps:
1. Create instances of OnlineSalesDTO with sample data to simulate online sales.
2. Mock the paymentInfoService.viewOnlineSales method to return a list of these OnlineSalesDTO instances.
3. Perform a GET request to the /ppms/payment/getOnlineSales endpoint.
4. Assert that the HTTP response status is OK (200).
5. Assert that the response content is correctly formatted as JSON and matches the expected list of online sales data.

Expected Inputs: None directly provided to the endpoint; data is fetched from service layer.

Expected Outputs:
1. An HTTP status of 200 (OK).
2. A JSON array representing online sales, expected to match the mocked data from paymentInfoService.viewOnlineSales.

Test Case 3: testGetInventoryByStatus

Purpose: This test case is designed to verify the functionality of the getInventoryByStatus method in retrieving inventory details based on stock status. It tests the method's ability to correctly format and return inventory data as JSON.

Functionality Under Test: The test evaluates whether the getInventoryByStatus method can accurately fetch a list of inventory items with their respective statuses and quantities, and if it can return this data properly in the response body as JSON. It also confirms that the correct HTTP status is returned.

Test Steps:
1. Define instances of InventoryStatusDTO with predefined stock status and quantity data.
2. Mock the employeeService.getInventoryDetailsByStockStatus to return a list of these InventoryStatusDTO instances.
3. Perform a GET request to the /ppms/employee/getInventoryByStatus endpoint.
4. Assert that the HTTP response status is OK (200).
5. Assert that the response content matches the expected JSON output, accurately reflecting the mocked inventory data.

Expected Inputs: None directly provided to the endpoint; the data retrieval is handled internally based on predefined logic in the service layer.

Expected Outputs:
1. An HTTP status of 200 (OK).
2. A JSON formatted response containing a list of inventory items, mirroring the data structure provided by the mocked service layer.

Test Case 4: testGetInventoryByStatusWhenException

Purpose: This test case aims to verify the robustness of the getInventoryByStatus method when handling exceptions, specifically testing its ability to manage unexpected backend errors like a database failure.

Functionality Under Test: The test assesses whether the getInventoryByStatus method correctly handles exceptions by returning an appropriate HTTP status code when an error occurs during the retrieval of inventory data.

Test Steps:
1. Mock the employeeService.getInventoryDetailsByStockStatus to throw a RuntimeException with a message indicating a "Database error".
2. Perform a GET request to the /ppms/employee/getInventoryByStatus endpoint.
3. Assert that the HTTP response status is BadRequest (400) to reflect the error condition adequately.

Expected Inputs: None directly provided to the endpoint; the error scenario is triggered by the mocked service behavior.

Expected Outputs:
1. An HTTP status of 400 (BadRequest), indicating that the server could not process the request due to an internal server error, such as a database failure.

System Tests:

Test Case 5: testCompleteAuthenticationFlow

Purpose: This test case is designed to validate the complete authentication flow, including user registration, initial login, and OTP verification, ensuring each step adheres to expected security and functional protocols.

Functionality Under Test:
Step 1: User Signup: Validates that new users can register successfully.
Step 2: Initial Login: Checks the initial login process where username and password are verified.
Step 3: OTP Verification: Tests the OTP verification process as the final step of user authentication.
Test Steps:

User Signup:
1. Create a User object with necessary details.
2. Perform a POST request to /ppms/user/signup with the user data serialized into JSON.
3. Assert that the HTTP response status is OK (200).

Initial Login:
1. Perform a POST request to /ppms/user/login/{username}/{password} using the registered user's credentials.
2. Assert that the HTTP response status is OK (200).

3. Assume a method exists to fetch or mock the OTP value necessary for the next step of authentication.

OTP Verification:

1. Mock the response of loginService.loginUserSecondStep to return a success message upon correct OTP verification.
2. Perform a POST request to /ppms/user/login/{id}/verifyotp/{otp} with the user ID and OTP.
3. Assert that the HTTP response status is OK (200) and the response content matches the expected success message "Login successful".

Expected Inputs:

1. **User Signup:** Includes username, email, and password.
2. **Initial Login:** Username and password of the registered user.
3. **OTP Verification**: User ID and OTP value.

Expected Outputs:

1. Successful HTTP responses (status code 200) for all steps.
2. A success message confirming OTP verification.

Test Case 6: testInventoryProductAssistantWorkflow

Purpose: This test case validates the end-to-end workflow of the inventory management system, ensuring the system correctly handles inventory status checks, updates to inventory, and assists customers through an employee assistant.

Functionality Under Test:

Step 1: Check Inventory Status: Tests whether the system can accurately display inventory statuses.
Step 2: Add or Update a Product in Inventory: Verifies the system's capability to update or add new products to the inventory.
Step 3: Assist Customers: Confirms the system can identify and provide details of an assistant employee to help customers.

Test Steps:

Check Inventory Status:

Mock the employeeService.getInventoryDetailsByStockStatus to return a list of InventoryStatusDTO.
Perform a GET request to /ppms/employee/getInventoryByStatus.
Assert that the HTTP response status is OK (200) and the content matches the expected JSON formatted inventory list.

Add or Update a Product in Inventory:

Mock the employeeService.addOrUpdateProduct to return a success message.
Perform a POST request to /ppms/employee/addOrUpdateProduct with productId and quantity as parameters.
Assert that the HTTP response status is OK (200) and the content matches the expected success message.

Assist Customers:

Prepare and mock an EmployeeDTO representing an assistant employee.
Mock the employeeService.findAssistantEmployee to return the prepared EmployeeDTO.
Perform a GET request to /ppms/employee/getAssistant.
Assert that the HTTP response status is OK (200) and the content matches the expected JSON formatted details of the assistant employee.

Expected Inputs:

Step 1: No direct inputs; data is fetched from service.
Step 2: productId (Long), quantity (Integer).
Step 3: No direct inputs; data is fetched from service.

Expected Outputs:
Step 1: JSON formatted list of inventory statuses.
Step 2: Success message indicating product has been successfully added or updated.
Step 3: JSON formatted details of an assistant employee.

Test Case 7: testCompletePaymentAndTrackOrder

Purpose: This test case aims to validate the full workflow of making a payment and then tracking the resulting order to ensure the processes integrate smoothly and function as expected.

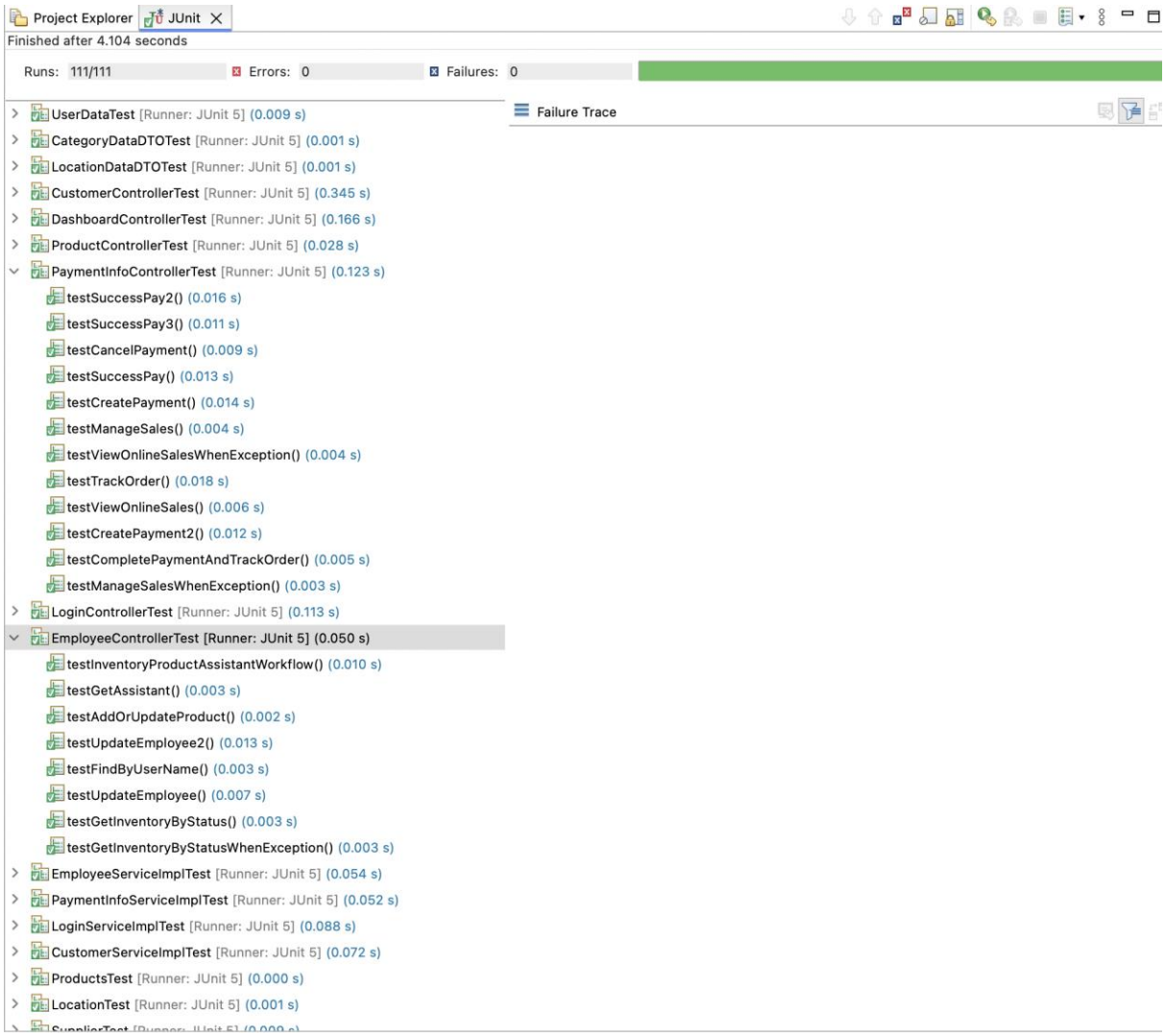
Functionality Under Test:
Step 1: Simulate a Successful Payment: Tests the system's ability to handle payment execution correctly.
Step 2: Track the Order After Successful Payment: Verifies that the order can be tracked using a unique tracking ID provided after the payment process.

Test Steps:
Simulate a Successful Payment:
Mock the paymentInfoService.executePayment method to return a new Payment object.
Perform a GET request to /ppms/payment/{userId}/success with placeholders for PayerID and paymentId.
Note: The assertion to check response status and content was missing in the initial setup and should be added to ensure that the payment is processed successfully.
Track the Order After Successful Payment:
Mock the paymentInfoService.getByOrderId method to return a PaymentInfo object based on a given tracking ID.
Perform a GET request to /ppms/payment/trackorderdetails/{trackingId}.
Assert that the HTTP response status is OK (200) and the response content is correctly formatted as JSON, reflecting the payment info data.

Expected Inputs:
Step 1: userId (String), PayerID (String), paymentId (String).
Step 2: trackingId (String), which is used to fetch the order details.

Expected Outputs:
Step 1: Successful processing of payment, ideally checked by asserting the HTTP status and the content to confirm the creation of a new payment entry.
Step 2: JSON formatted PaymentInfo reflecting details of the tracked order, confirming that the order details are accessible and correctly presented.

All the test cases are running successfully and the above mentioned test cases are written in Payment Controller Test, Employee Controller Test and Customer Controller Test and Login Controller Test. Below is the execution snapshot of all the test cases.

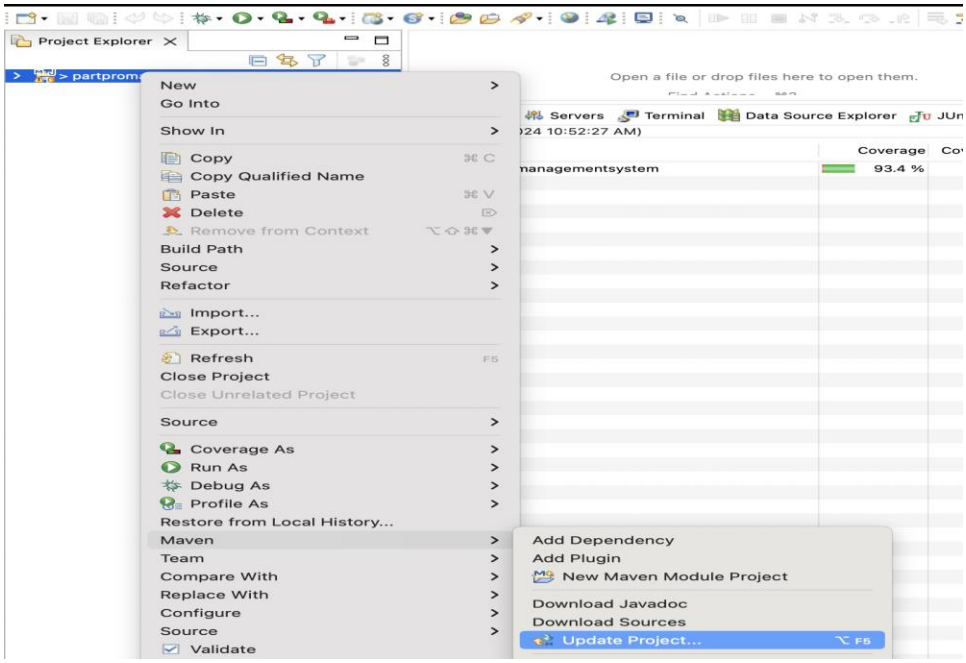


D. Report Compilation Instructions:

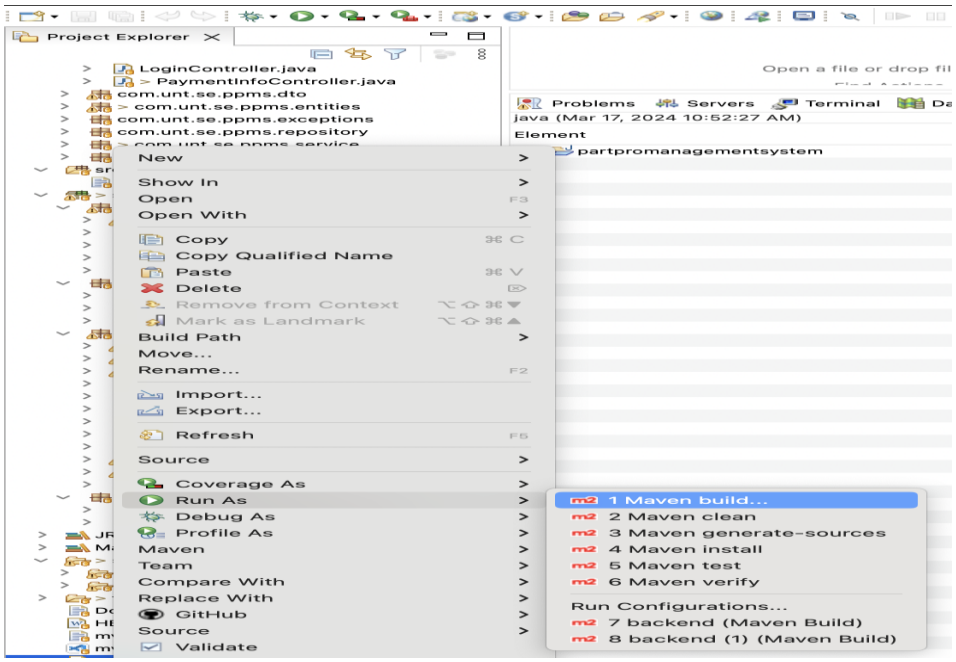
For Backend:

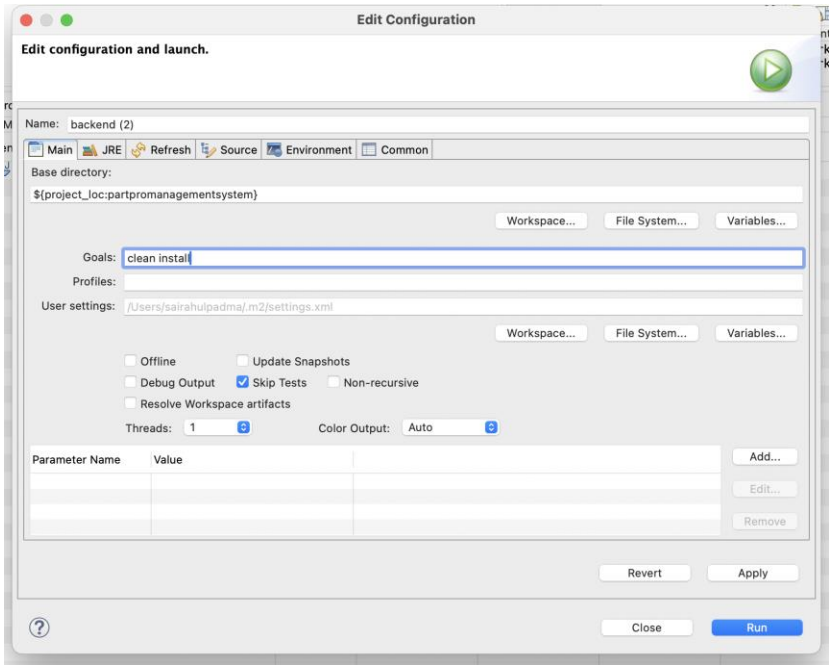
1. Install eclipse from the below link. If the OS is of windows, install it from below link.
   https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2024-03/R/eclipse-jee-2024-03-R-win32-x86_64.zip
   If the OS is of Mac, install is from below link.
   https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2024-03/R/eclipse-jee-2024-03-R-macosx-cocoa-aarch64.dmg
2. Install the java development kit for getting java environment. Install it from below link.
   https://www.oracle.com/java/technologies/javase/jdk21-archive-downloads.html
   Install java of version 21.0.2 based on OS type.
3. Install MySQL workbench from the below link.
   https://dev.mysql.com/downloads/workbench/
   Install MySQL Workbench 8.0.36 based on OS type.

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



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

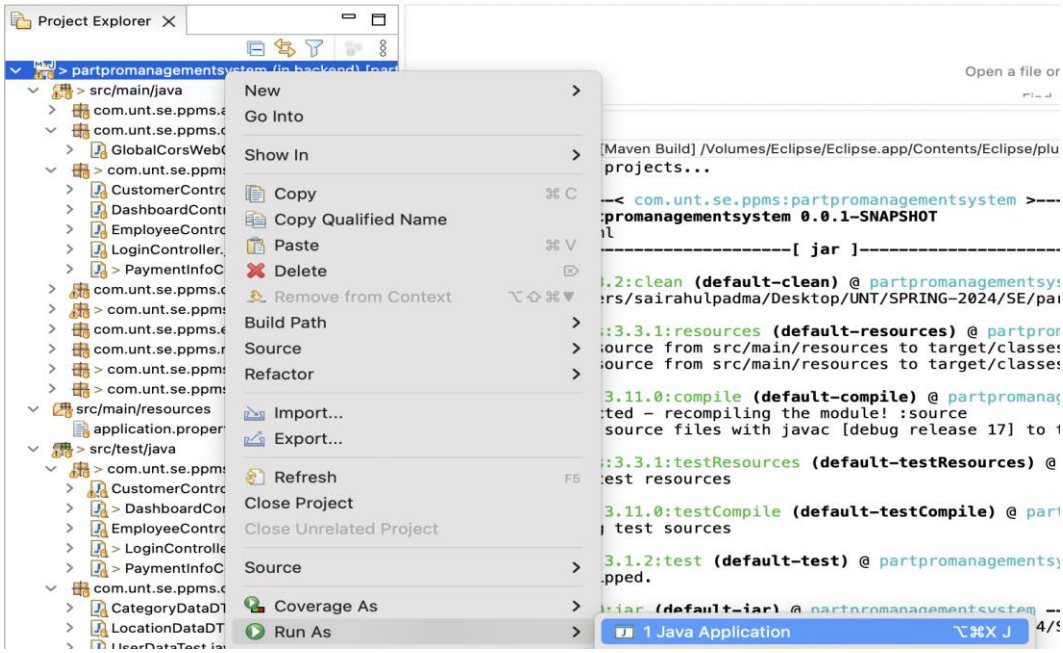




6. You would get the below output in the console.

```
[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] ------------------------------------------------------------------------
```

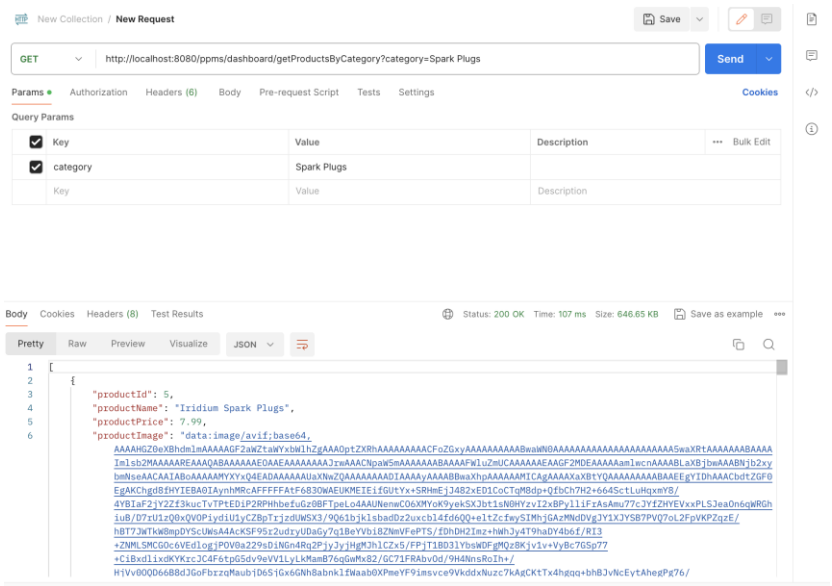
7. Update the username and password of your database credentials in application.properties which is resided in src/main/resources folder.
8. Run the java application once the application is compiled successfully.



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

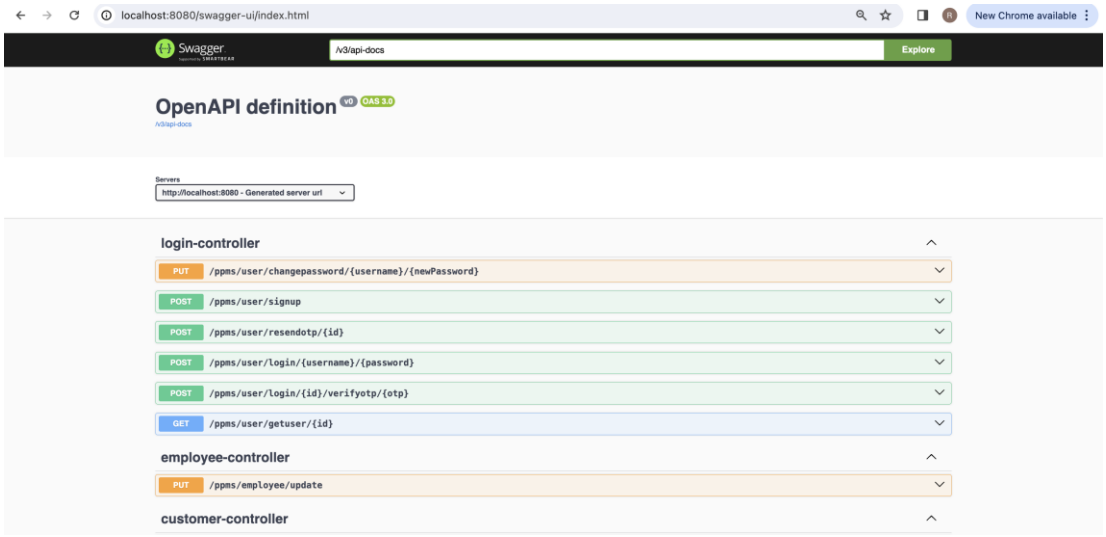
```
2024-03-17T11:56:01.863-05:00  INFO 29987 --- [  restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2024-03-17T11:56:02.073-05:00  INFO 29987 --- [  restartedMain] o.s.d.j.r.query.QueryEnhancerFactory     : Hibernate is in 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
```

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



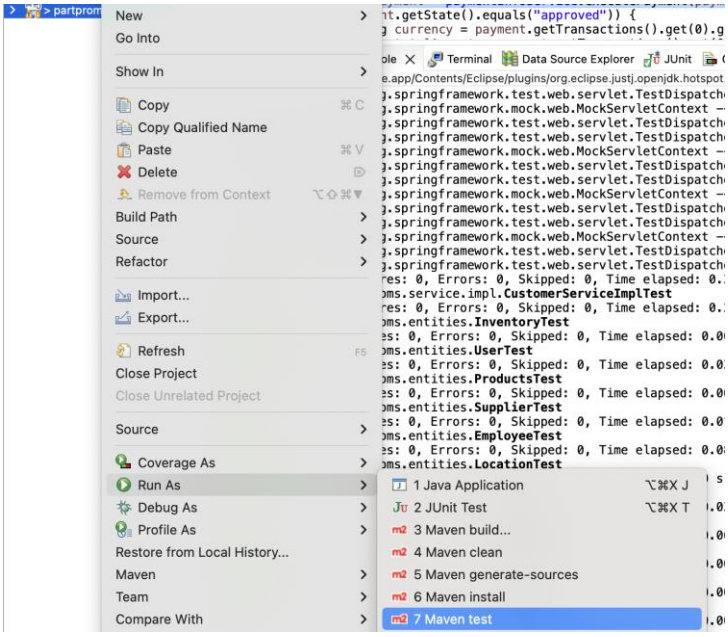
11. If you need all the API collection, Run the swagger link in your browser. We need to run the application before opening Swagger URL.

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

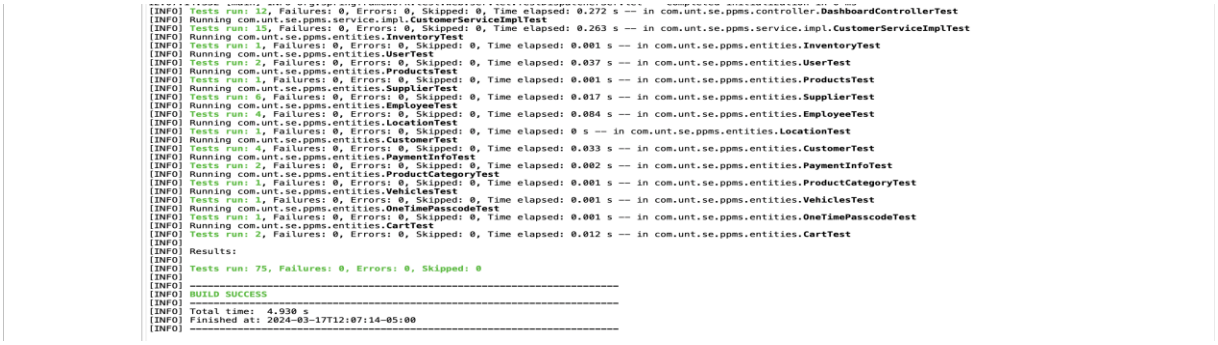


For Backend Test cases:

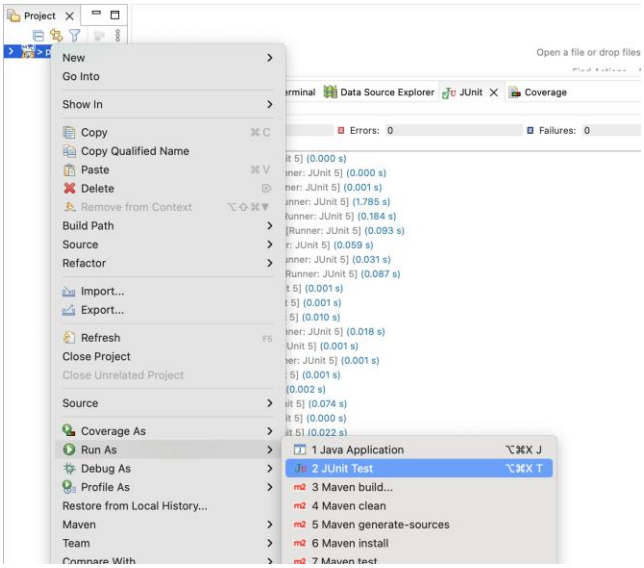
1. Compile the junit tests which were in src/test/java. Compile the test cases as shown below.



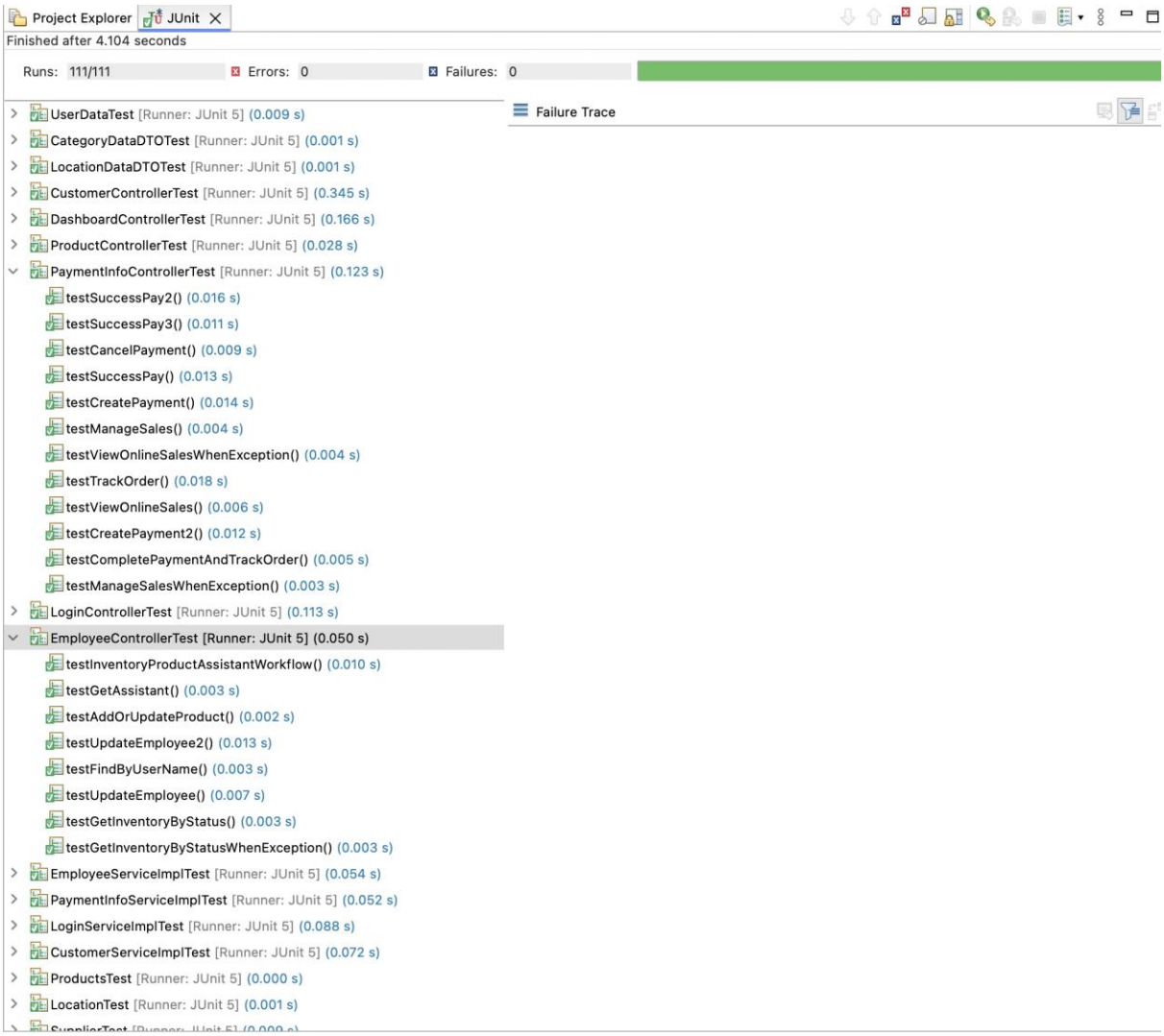
You would get the below output from console.



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

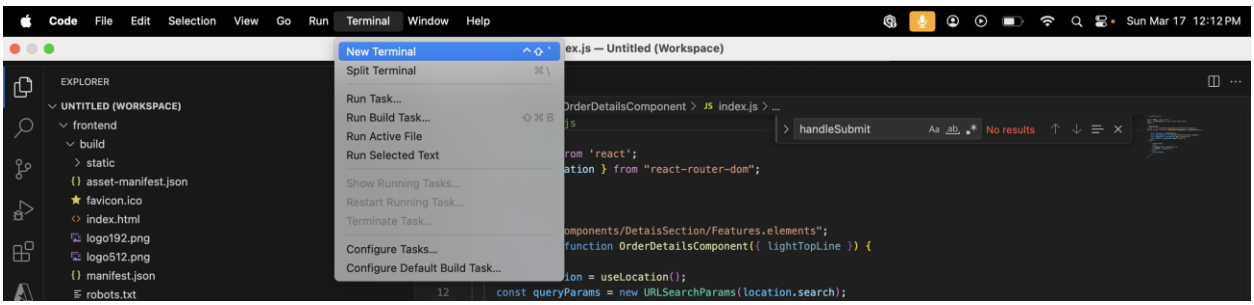


You would get the below output from the console.



For Front end:

1. Install Node JS of version 14.x.x from below link based on OS type.
   https://nodejs.org/en/blog/release/v14.17.3
2. Install visual studio code from below link based on OS type.
   https://code.visualstudio.com/download
3. Open the terminal in vscode as shown below.



4. 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
```

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

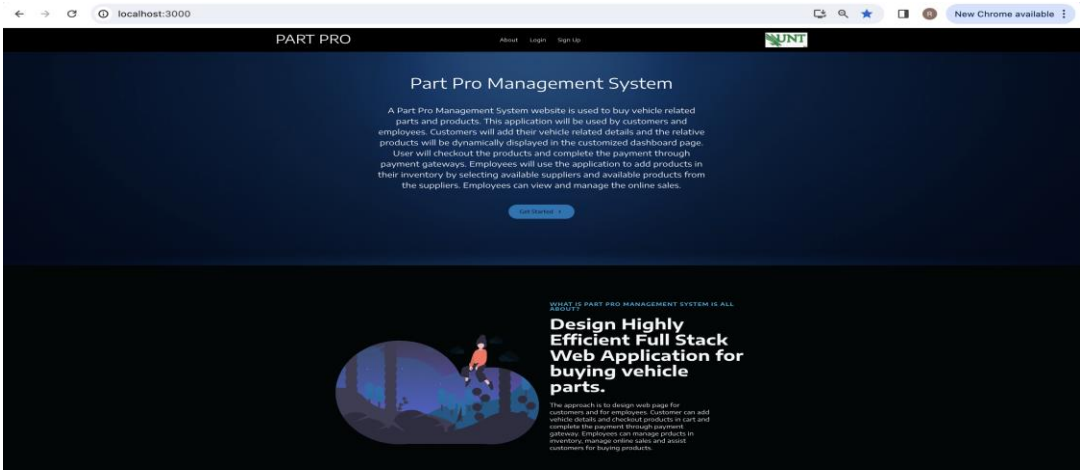
```
The build folder is ready to be deployed.
You may serve it with a static server:

  npm install -g serve
  serve -s build

Find out more about deployment here:

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

6. 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.



E. **Report User Manual**: 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.

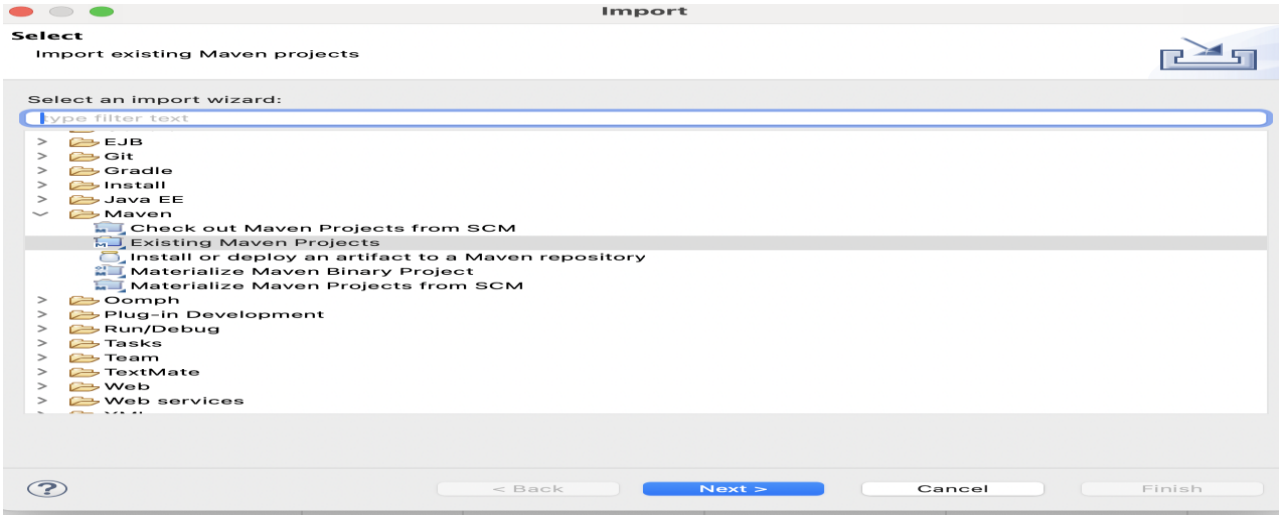
12. . If the OS is of windows, install eclipse from below link.
    https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2024-03/R/eclipse-jee-2024-03-R-win32-x86_64.zip
    If the OS is of Mac, install eclipse from below link.
    https://www.eclipse.org/downloads/download.php?file=/technology/epp/downloads/release/2024-03/R/eclipse-jee-2024-03-R-macosx-cocoa-aarch64.dmg

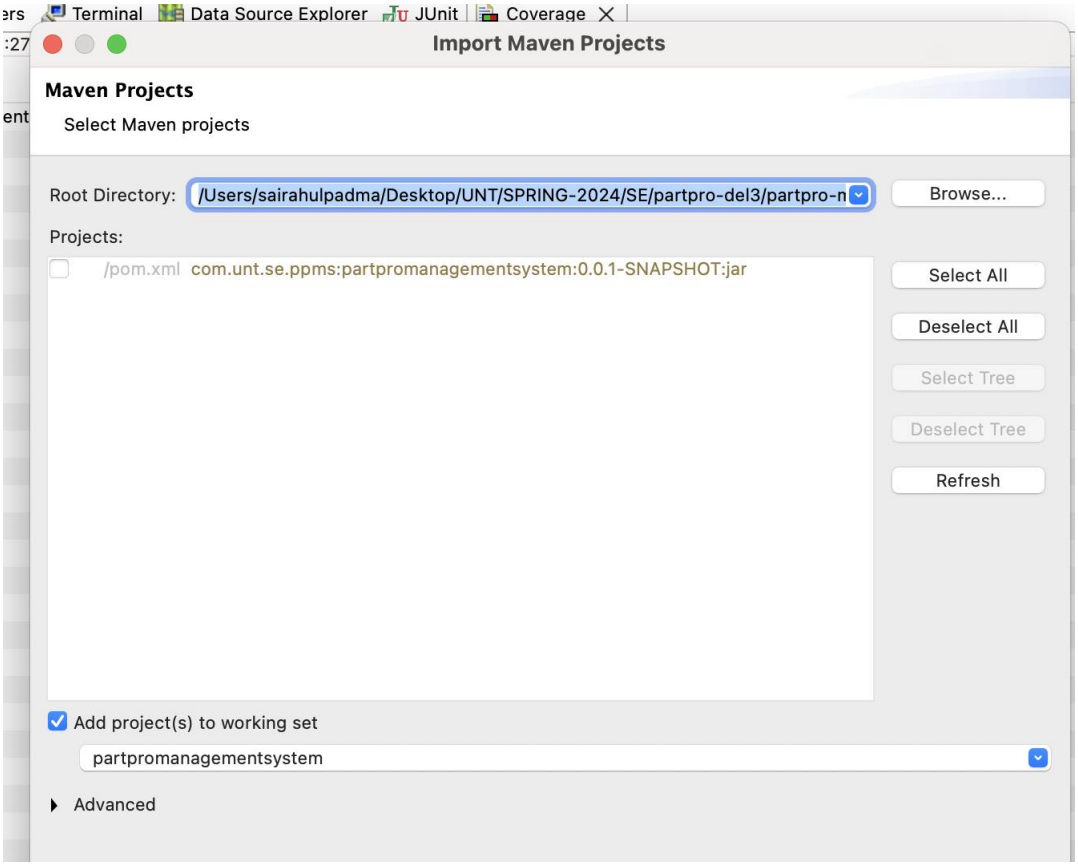
MySQL Workbench Installation: https://dev.mysql.com/downloads/workbench/
Install MySQL Workbench 8.0.36 based on OS type.
Postman Installation: https://www.postman.com/downloads/

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
   Install java of version 21.0.2 based on OS type.
6. After successful installation of jdk verify the java version through terminal. And the commands are java –version and javac –version

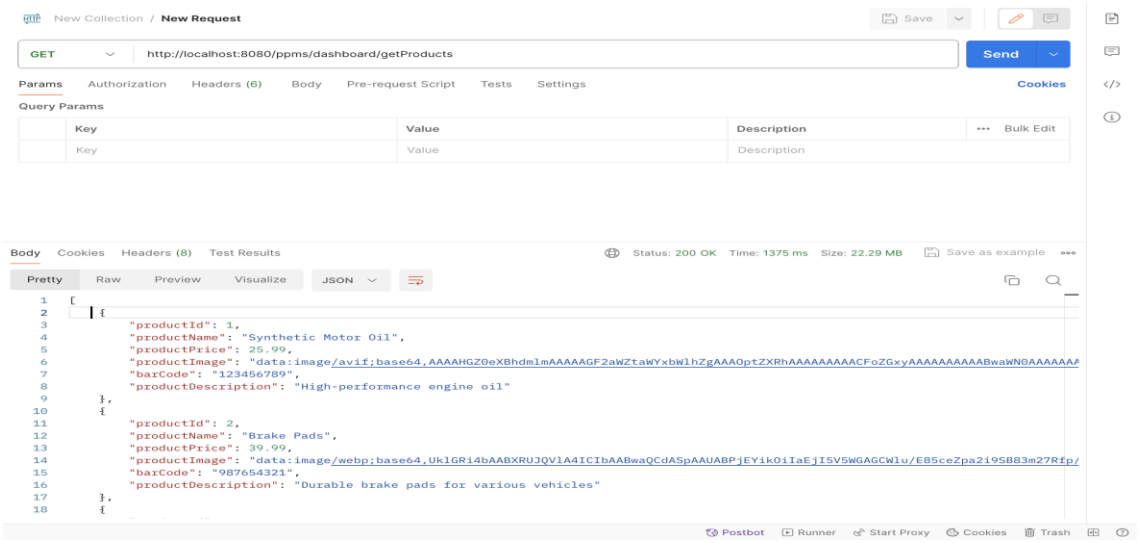
```
sairahulpadma@Sais-MacBook-Air-4 ~ % java --version
java 21.0.2 2024-01-16 LTS
Java(TM) SE Runtime Environment (build 21.0.2+13-LTS-58)
Java HotSpot(TM) 64-Bit Server VM (build 21.0.2+13-LTS-58, mixed mode, sharing)
sairahulpadma@Sais-MacBook-Air-4 ~ % javac --version
javac 21.0.2
sairahulpadma@Sais-MacBook-Air-4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

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.





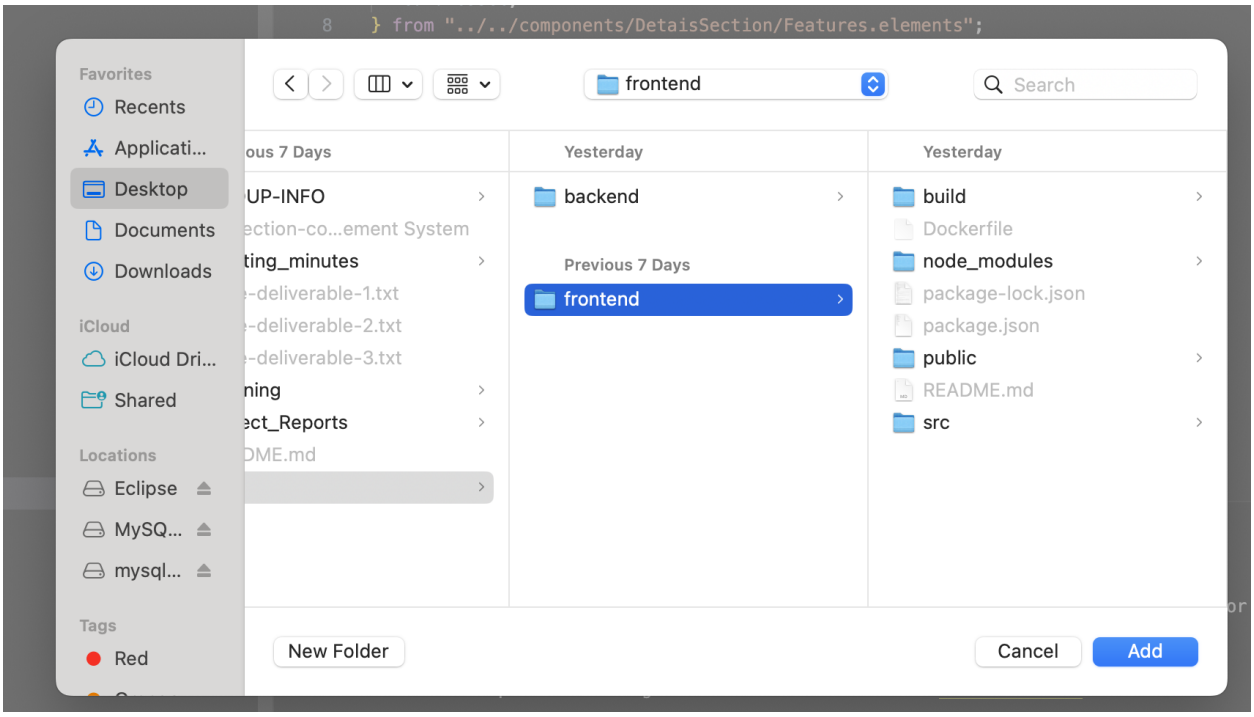
8. Once the application is up and running test the APIs through postman. We can also test all the API through swagger. The swagger is used to store all the API collection and this configuration is written in the back-end code. The swagger link would be running once the application is built successfully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



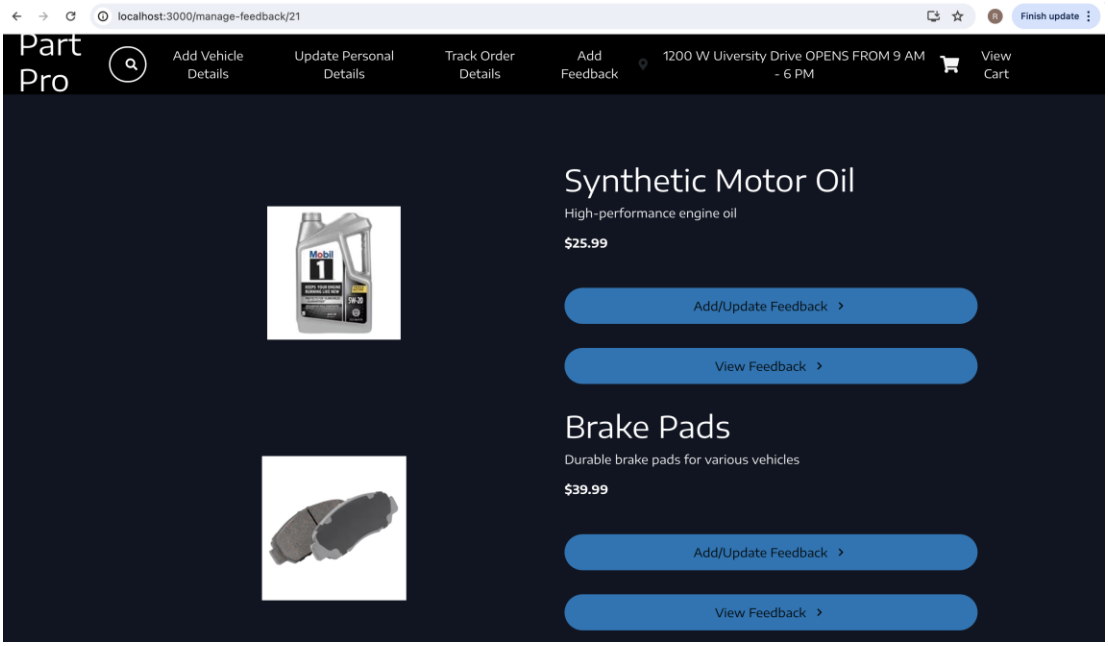
10. Now install the front-end software's for working on front end code. Install the Node JS for running front end on web server and it installs all required dependencies. Install Visual studio code for front end code development.
    Node JS Installation: https://nodejs.org/en/blog/release/v14.17.3
    Install Node JS of 14.7.3 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 ~ %
```

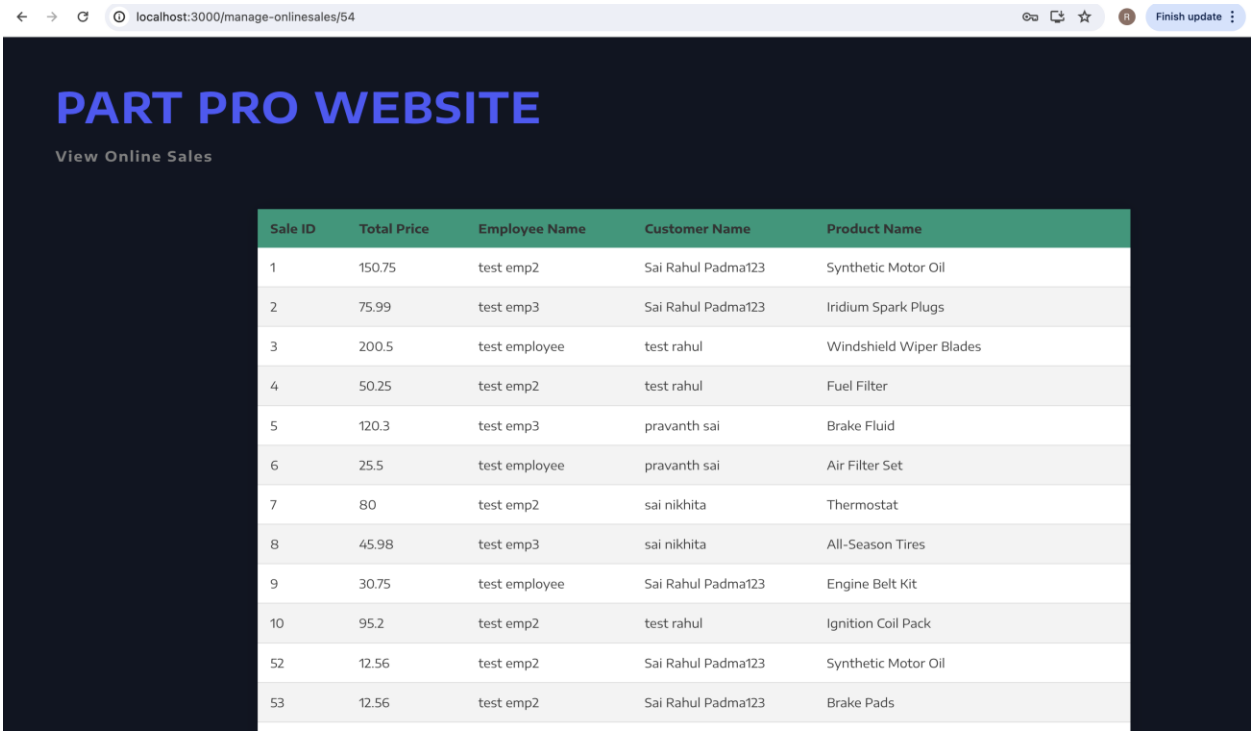
12. Go to the directory till src/frontend and import the front-end repository into vs code.
    Click on File→ Add Folder to Workspace
    Add the root directory till src/frontend and click Add. The front-end code is imported successfully.



13. After the application is up and running use the application through web link. The link is pasted below.
    http://localhost:3000/
14. Once customer lands on dashboard screen, customer can view add feedback option on navbar. Once customer clicks on add feedback button, customer would land on add feedback screen. And the below figure is the feedback screen.

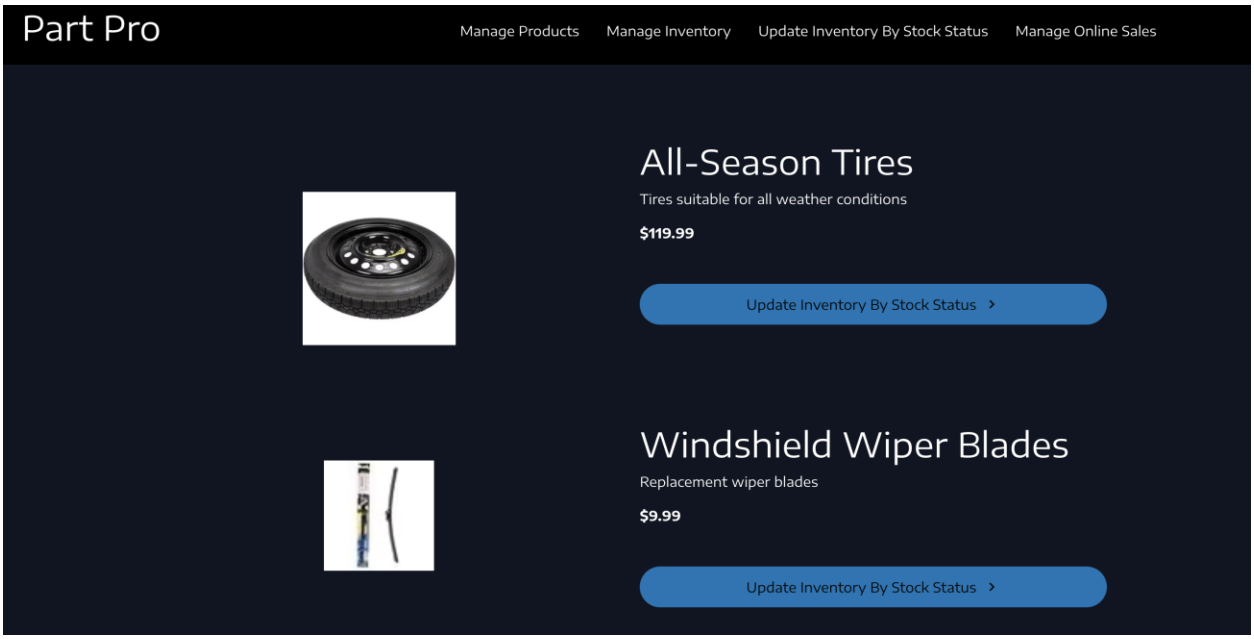


15. Once employee lands on dashboard screen, employee would click on manage online sales option on navbar. Employee would land on view online sales screen. And the below figure is the online sales screen.



Sale ID	Total Price	Employee Name	Customer Name	Product Name
1	150.75	test emp2	Sai Rahul Padma123	Synthetic Motor Oil
2	75.99	test emp3	Sai Rahul Padma123	Iridium Spark Plugs
3	200.5	test employee	test rahul	Windshield Wiper Blades
4	50.25	test emp2	test rahul	Fuel Filter
5	120.3	test emp3	pravanth sai	Brake Fluid
6	25.5	test employee	pravanth sai	Air Filter Set
7	80	test emp2	sai nikhita	Thermostat
8	45.98	test emp3	sai nikhita	All-Season Tires
9	30.75	test employee	Sai Rahul Padma123	Engine Belt Kit
10	95.2	test emp2	test rahul	Ignition Coil Pack
52	12.56	test emp2	Sai Rahul Padma123	Synthetic Motor Oil
53	12.56	test emp2	Sai Rahul Padma123	Brake Pads

16. Once employee lands on dashboard screen, employee would click on update inventory by stock status option on navbar. Employee would land on Update Inventory Levels by stock status screen. And the figure shared below.



F. Report Ending Feature Summary:

Implemented Features:

1. Successfully implemented Authentication feature. Authentication feature includes user registration, login using username and password, login using MFA, forgot password and change password.
2. Successfully implemented Dashboard feature. Dashboard feature includes get all products, get all categories, add vehicle details, update personal details, track order details, and add feedback.
3. Successfully implemented Payment features. Payment feature contains add products to cart, update products in cart, remove products in cart, View cart details, complete payment, cancel payment and view order details.
4. Successfully implemented Employee features. Employee feature contains manage online sales, add products, update products, delete products, assist customers for in-store pickup, add products to inventory, update products to inventory and update inventory based on stock status.

Limitations:

1. Search Functionality: Parts pro must have efficient search functionality. Users may find it more difficult to locate the parts they need fast if search algorithms or filter options are limited.
2. Customer Service Integration: Integrating efficient online customer service features, such as live chat and quick response to inquiries can be implemented.
3. Mobile Experience: Part pro is not compatible for mobile experience. It is not implemented for mobile experience. It is limited to web channel.
4. Integration with Physical Stores: Seamless integration between the online portal and physical store operations can be challenging.

Unimplemented Features:

1. Recommendation Engine: A function that makes product recommendations to customers depending on their search habits, past purchases, and browsing history. This could personalize product offerings and improve the purchasing experience.
2. Loyalty and Rewards Program: Establishment of a loyalty program in which consumers can exchange their points for discounts or exclusive offers after making purchases. This characteristic boosts consumer engagement and promotes repeat business.
3. Live Chat Support: Integration of a live chat system for customer support that enables users to speak with agents right away for assistance with purchases, product details, or difficulties.
4. Advanced Analytics for Employees: An elaborate analytics dashboard to monitor client purchasing patterns, inventory turnover, and sales trends for staff members. Making well-informed decisions regarding marketing tactics and stock replenishment could benefit from this.

Future Plans:

1. User Profile Customization: To create a more personalized buying experience for customer who can add vehicle preferences, past purchase history, and other details to their accounts.
2. Wish List Feature: Introduce a wish list where customers can save items they are interested in purchasing later. This can also be used to notify users of price drops or stock availability.
3. Subscription Services: Provide subscription-based services for consumable goods like motor oil or wipers for windshields. Consumers can choose to get these products on a regular basis, so they never run out.
4. Order History with Reorder Option: Give customers access to an extensive order history log together with the practicality of a single-click reorder option for products they've already bought.

G. Report Reflection:

Accomplishments:

1. We have implemented all features which are planned for development phase 3.
2. We have done end to end system testing by including multiple components within single test.
3. We have worked on couple of non-functional requirements like performance, optimization, and scalability.
4. We have designed and successfully developed feedback module which is a suggestion given during workshops.
5. We have successfully deployed code for all the phases. Created load balancer for enhanced security.

Positive Aspects which went well:

1. We've established robust version control protocols, ensuring that our codebase is well-managed and that changes are documented.
2. Followed agile processes has allowed us to guarantee regular iterations and ongoing feedback.
3. We've maintained clear and effective communication channels within our team.
4. Unit tests, integration tests, and system tests are all part of our strategic testing approach, which has helped us identify issues early and maintain excellent code quality as we've developed.
5. We implemented frequent knowledge-sharing workshops that have improved our team's overall productivity and skill set.

Areas of Improvement:

1. Building a more thorough framework for automated testing can help identify problems early on, cut down on manual testing, and provide more reliable quality control.

2. A more consistent and interesting user experience across all platforms can be achieved by optimizing the application's responsiveness and layout for a range of devices and screen resolutions.
3. Reducing the learning curve and increasing overall satisfaction can be achieved by making navigation simpler and user flows more efficient.

H. Report Member Contribution Table:

Member name	Contribution description	Overall Contribution (%)	Note (if applicable)
Sai Rahul Padma	I have worked on code deployment inside azure VM for phase 3 code, design of Use case, sequence , class diagrams, required database scripts execution. I have worked on integration and system tests.	14.5	
Nikhita Muvva	I have worked on manage online sales related pages design, routes to manage online sales, service API integrations and required component and CSS designs.	12.5	
Sai Samyuktha Paspuleti	I have worked on back-end manage online sales related API development and unit test cases, exception handling and service implementation.	12	
Rishika Yalamanchili	I have worked on front end feedback related pages design, routes to product pages, service API integrations and required component and CSS designs.	12	
Pavani Venigalla	I have worked on back-end feedback related API development, unit test cases, exception handling and service implementation.	12.5	
Himabindu Chunduri	I have worked on front end update inventory by stock status related pages design, routes to product pages, service API integrations and	12	

	required component and CSS designs.		
Sneha Reddy Gangannagari	I have worked on back-end update inventory by stock status related API development, unit test cases, exception handling and service implementation.	12.5	
Jhasha Sri Ede	I have worked on front end changes for employee navbar and customer navbar. Worked on creating API integration in services folder. Added new table style components CSS which is used for view online sales component.	12	