THE UNIVERSITY OF SYDNEY

School of Information Technologies
Faculty of Engineering & IT

ASSIGNMENT/PROJECT COVERSHEET - GROUP ASSESSMENT

Unit of Study: SOFT2412

Assignment name: Tools for Agile Software Development.

Tutorial time: Tuesday 12:00pm 2:00pm **Tutor name:** Hunter

DECLARATION
We the undersigned declare that we have read and understood the *University of Sydney Student Plagiarism: Coursework Policy and Procedure*, and except where specifically acknowledged, the work contained in this assignment/project is our own work, and has not been copied from other sources or been previously submitted for award or assessment.

We understand that understand that failure to comply with the *Student Plagiarism: Coursework Policy and Procedure* can lead to severe penalties as outlined under Chapter 8 of the *University of Sydney By-Law 1999* (as amended). These penalties may be imposed in cases where any significant portion of my submitted work has been copied without proper acknowledgement from other sources, including published works, the internet, existing programs, the work of other students, or work previously submitted for other awards or assessments.

We realise that we may be asked to identify those portions of the work contributed by each of us and required to demonstrate our individual knowledge of the relevant material by answering oral questions or by undertaking supplementary work, either written or in the laboratory, in order to arrive at the final assessment mark.

Project team members				
Student name	**Student ID**	**Participated**	**Agree to share**	**Signature**
1. Yi Fei (Fiona) Chen	510430519	(Yes) / No	(Yes) / No	YI FEI CHEN.
2. Callista Hardi	530297639	(Yes) / No	(Yes) / No	Callista
3. chunfi han	510045254	(Yes) / No	(Yes) / No	chunfi han
4. Yilei Ma	530146902	(Yes) / No	(Yes) / No	Jennifer Ma
5.		Yes / No	Yes / No	
6.		Yes / No	Yes / No	
7.		Yes / No	Yes / No	
8.		Yes / No	Yes / No	
9.		Yes / No	Yes / No	
10.		Yes / No	Yes / No	

Level 2, SIT Building, J12
The University of Sydney
NSW 2006 Australia

T +61 2 9351 3423
F +61 2 9351 3838
E sit.info@sydney.edu.au
sydney.edu.au/it

ABN 15 211 513 464
CRICOS 00026A

1. Group Collaboration

1.1 Individual Contribution

SID	**contribution**
510430519	● Administrative works ○ schedule meetings, record meeting minutes, breakdown spec into tasks, allocate tasks to members, and track progress ● Code ○ JsonManagement and Order Class ○ Bug fixing for the final integration in App ○ Junit testing for the responsible classes ○ Set up Jenkins, github and ngrok ● Report ○ Jenkins, Junit and refinement in README. ○ Creation of project board and providing evidence for conflict resolution for GitHub section ○ Additional gradle task for Gradle section
530297639	● Code ○ User Class ○ OrderHistory Class ○ Junit testing for User ○ Junit testing for OrderHistory Class ○ Bug fixing for the responsible classes ● Report ○ GitHub ○ JUnit test cases for User and Order History class
510045254	● Code ○ Menu (Admin) Class ○ Bug fixing for the responsible classes ○ Junit testing for the Menu Admin ● Report ○ Application development report ○ Readme
530146902	● Code ○ Cart, App and Menu (General) Class ○ Bug fixing for the responsible classes ○ Junit testing for the Menu Admin and General Class ○ Junit testing for the Cart Class ○ Add dependencies required for responsible code in build.gradle ● Report ○ Gradle ○ JUnit test cases for Cart, Menu Admin and Menu General Class

1.2 Group Contribution

Below are our team contracts prior to the team formation, with the group contribution to the software development process for this project:

1. Every member must attend all meetings scheduled per week as the meeting is scheduled beforehand according to everyone's availability.

2. Every member should contribute to the planning and implementation of the project.

3. Use version control systems like Git and platforms like GitHub or GitLab to manage the project's source code. We control that everyone's Gradle and Java versions are consistent. This allows for easy collaboration and tracking of changes made by different group members.

4. Employ task management tools or methodologies like Kanban or Scrum to keep track of tasks, assign them to group members, and monitor progress.

5. Weekly group meetings and mini-meetings to allow members to update on the progress, challenges, and planning for the next stage of the development process.

6. Maintain project documentation is accessible and available to every members, including meeting minutes, marking criteria, assignment specification requirements, and project design

7. Assign group members to perform testing and quality assurance to identify and resolve issues and ensure the application functions correctly.

1.3 Group Communication

2023.08.24 Meeting Minutes , 2023.08.28 Meeting Minute ,
2023.08.29 Meeting Minutes , 2023.09.01 Meeting Minutes ,
2023.09.05 Meeting Minutes , 2023.09.07 Meeting Minutes

1. https://docs.google.com/document/d/1nXK0EbYmLFuDgXvPRRvEVnbsJQgfl315c3L_sizrMFc/edit?usp=drive_link
2. https://docs.google.com/document/d/1AcPbgIJdIZ-6OdkIyknmzC8t7hoRRyHhqGD5X4q3kYc/edit?usp=drive_link
3. https://docs.google.com/document/d/1Co12B3JEpLYoy4Q_9jSAzpLJG5C__d-JnniS0n7F0PI/edit?usp=drive_link
4. https://docs.google.com/document/d/13DLgh71gswgGNsCAP2Mk3i6Uxn8Mpj9Rd-gx3NzONOQ/edit?usp=drive_link

5. https://docs.google.com/document/d/1DkAdqry4-3W-TSI6DFJq1UNkjjjm51CvM2bME2rU0I8/edit?usp=drive_link

2. README file

In the project, we made sure to include a README file. Within the README file, we provided the project overview and explained its features in detail. We also included instructions on how to run and test the program.

README.md

Lab02-Hunter-Group4-A1

Overview

This is a basic order management application. Order data is stored in a JSON file and manipulated through a Java interface. The application is designed to accommodate two types of users: guests and administrators.

Features

For general users:

1. **View Menu Categories:** This feature allows customers to view all available categories on the menu.
2. **View Menu Details:** This feature allows customers to select a category and view detailed information of all items within that category, including the item name, description, and price.

For admin users:

1. **Add Menu Items:** This feature allows administrators to add new items to a specific category. They need to provide the category, item name, item description, and price.
2. **Update Existing Items:** This feature allows administrators to update the description and price of existing items in a category.
3. **Delete Items:** This feature allows administrators to delete items from a category.
4. **Edit Categories:** This feature allows administrators to change the name of existing categories.
5. **View Order History:** Administrators can view all past order history.
6. **Query Specific Order:** Administrators can query an order with a specific order number.
7. **Register New Admins:** Users who are already administrators can register new admin users.
8. **Login:** Users can login using their username and password.
9. **Logout:** Users can logout of their account.

Fig 1. README Overview and Features

Environment Requirements

- Java version: 16.0.2
- Gradle version: 7.5.1
- Junit5

Fig 2. README Environment Requirements

Preparation

1. Open a terminal window
2. Type `gradle -v`
3. If Gradle is not installed, follow the instructions here.
4. Please use gradle with version at least 7.5.1. If you have an older version, please update it by the following command:

```
gradle wrapper --gradle-version 7.5.1
```

5. Open a new project in text editor (e.g. VS Code or IntelliJ) with Gradle support.
6. Perform gradle init to initialise the project.

```
gradle init
```

7. Clone the repository to your local machine.

```
git clone https://github.sydney.edu.au/SOFT2412-COMP9412-2023S2/Lab02-Hunter-Group4-A1.git
```

6. Check the gradle works properly by running the following command:

- Clean the cache of the project:

```
./gradlew clean
```

- Build the project:

```
./gradlew build
```

Fig 3. README Preparation

How to Run

This application is a Gradle project. To run the application interactively, navigate to the directory containing the `build.gradle` file and run the following command:

```
./gradlew runInteractive --console=plain
```

How to Test

Tests are written using JUnit. To run the tests, use the following command in the same directory as the `build.gradle` file:

```
./gradlew clean test
```

Dependencies

This application uses the following libraries:

- json-simple (version: 1.1.1): For parsing and manipulating the JSON file.
- Jackson (version: 2.12.3): For serialization and deserialization of JSON.

The Gradle build file should manage these dependencies automatically.

Fig 4. README How to Run and Test, Dependencies

How to Contribute

If you'd like to contribute to this project, please follow these steps:

1. Fork the repository.
2. Create a new branch in your forked repository.
3. Make your changes in this new branch.
4. Submit a pull request from the new branch in your forked repository to the main branch in the original repository.

Remember to write a clear and concise commit message and pull request description, explaining your changes and the reasons for them.

License

This application is open-source. You are free to use and modify it, but you must credit the original author and source.

Reference:

1. https://www.geeksforgeeks.org/how-to-convert-map-to-json-to-hashmap-in-java/
2. https://www.baeldung.com/java-convert-hashmap-to-json-object

Fig 5. README How to Contribute and License

3. GitHub

GitHub is a robust platform for version control and collaboration. In this project, GitHub was used to streamline our project workflow, ensuring everyone in the group has access and can edit, track changes and overall contribute to the project.

3.1 Setup

In order to use GitHub, we had to first set up GitHub for it to properly facilitate our work. This was done by following the guide provided on the Week 3 Tutorial Ed Lessons page.

The set up steps include creating a new project in GitHub, followed by establishing a new repository to store the project files. In order to get the project files onto our local, we used the git clone command. This would allow each member to have a copy of the project files to work on in their local.

Other than that, we also had to set up our local repositories before working on the project. This was done by using git config.

```
cd [path to your project]
git config --local user.name "<Your Name>"
git config --local user.email "<Your USYD Email Address>"
```

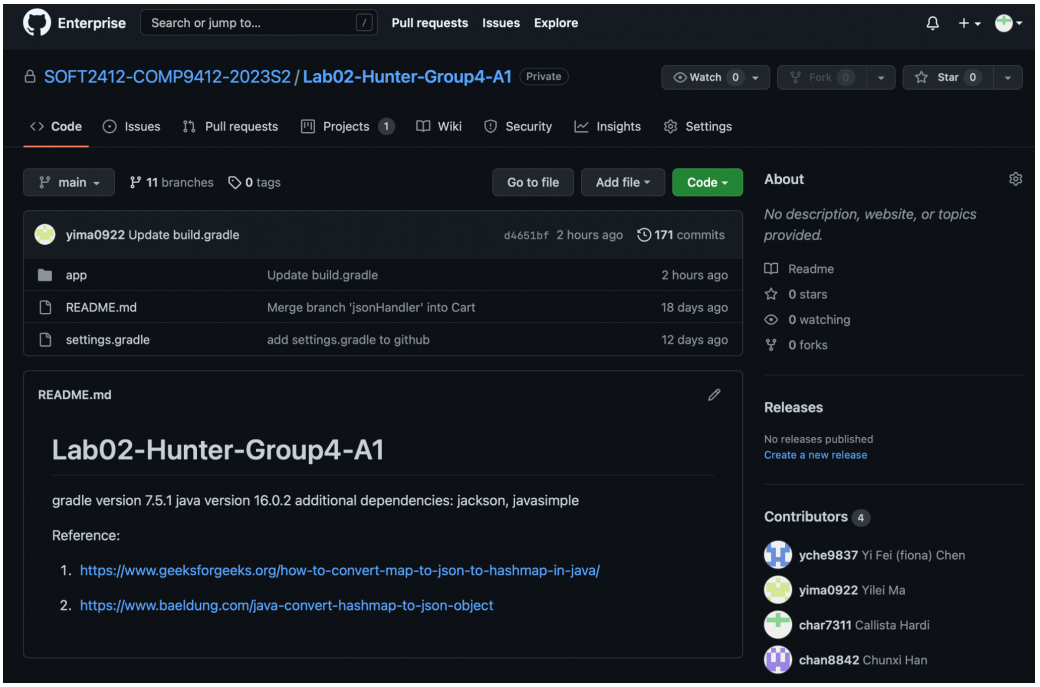


Fig 6. GitHub Project Repository

3.2 Git Commands

Some of the main git commands used during the project are:
Git add:
We use git add to stage our changes before committing them. We can specify which files to include in the next commit by using this git command, git add fileName.txt.

Git commit:
After a file is added to the staging area, we can record the changes in the file by committing it. The git commit command creates a snapshot of the staged files and a message associated with the changes made, git commit -m message. We use this command in the project to record the changes made to the repository.

Git commit -am:
This git command is a convenient way to stage and commit changes in only one step. Instead of using a separate git add and git commit command, we can skip the git add command by simply adding -am after git commit. This will automatically stage the modified files and commit them.

Git push:
The git push command was used to send our local commits to the remote project repository. This allows other group members to access any changes we made in our work.

Git fetch:
We use git fetch to retrieve any changes from the remote project repository without automatically merging them into our local branch. This command allows us to have control on how we want to integrate the changes into our branch, preventing any unexpected issues in our own code.

Git merge:
The git merge command was used whenever we wanted to combine changes from one branch into another branch.

Git pull:
Another way to fetch and merge changes would be to use the git pull command. By using the git pull command, we can fetch and merge in a single step. So instead of having git fetch and then git merge, we can simply use git pull. This is an efficient method to get the latest updates from the remote repository into our local.

Git config:
We used git config to help manage our git configuration settings. In the set up stage of the project, we had to set up our local repositories. This was done by using the git config command.

```
git config --local user.name "<Your Name>"
git config --local user.email "<Your USYD Email Address>"
```

Git checkout:
The command git checkout was used in the project whenever we wanted to switch branches. Other than that, we also used git checkout to access and extract new changes from a specific file.
`git checkout origin/main -- app/src/main/java/lab02/hunter/group4/a1/User.java`

Set Upstream:
We can add --set-upstream flag after git push to set the upstream branch for our local branch. This is to ensure Git knows where to push the changes to.
`git push --set-upstream origin orderHistory`
In this code, we updated the orderHistory branch in the remote repository with the changes made locally.

3.3 Pull Requests

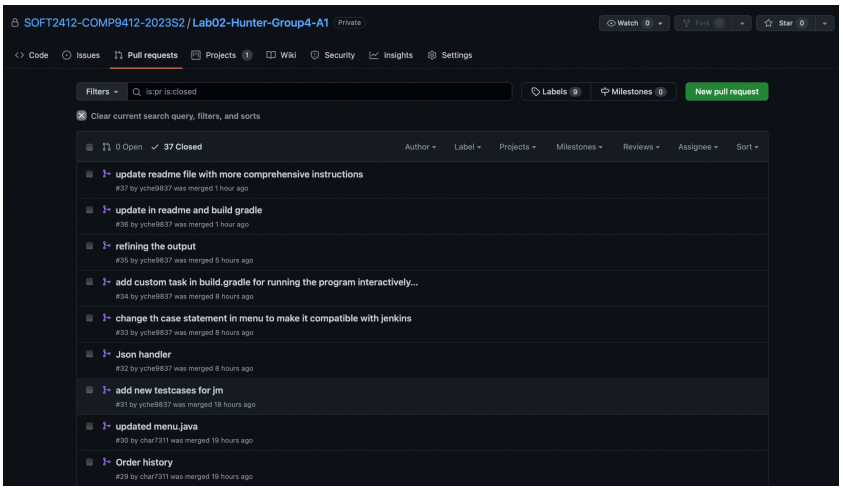


Fig 7. Closed Pull Requests

3.4 Branching

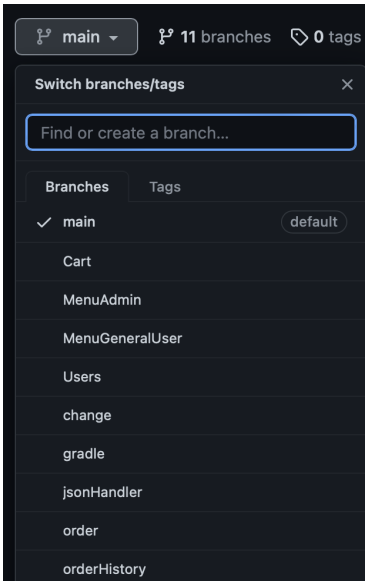


Fig 8. Branches in GitHub

3.5 Dealing with Issues

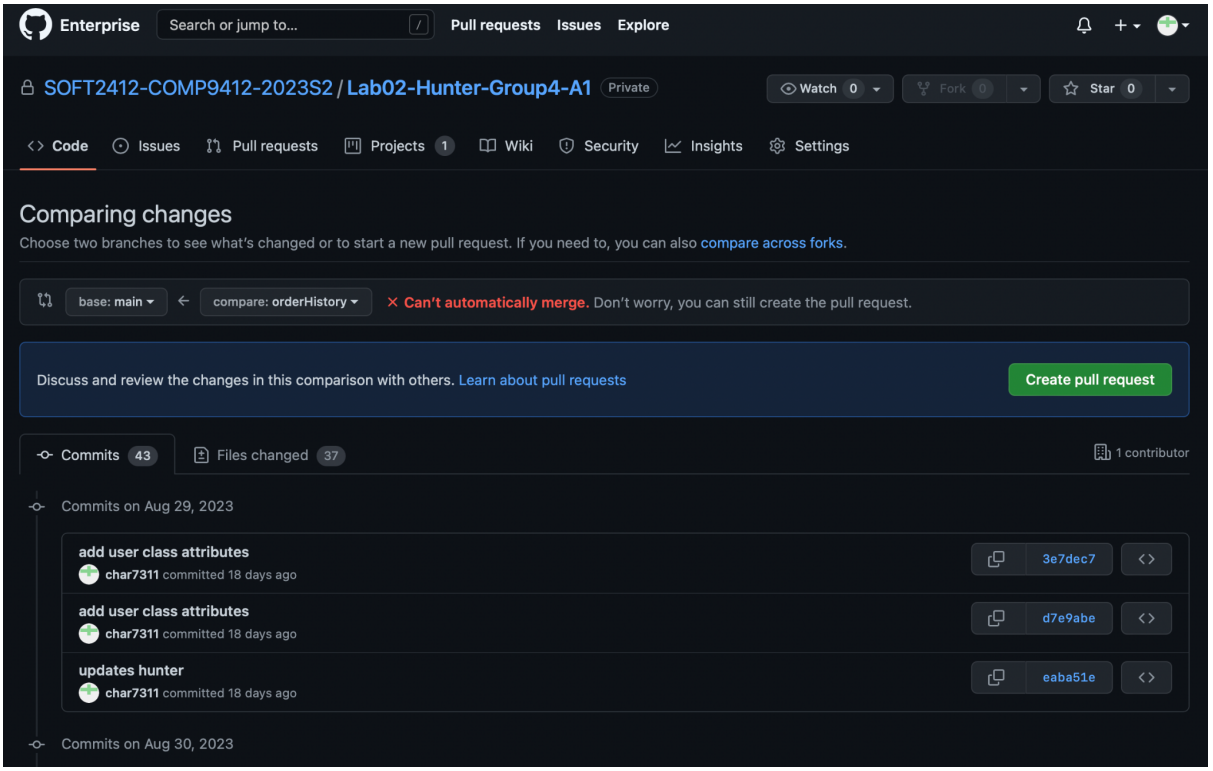


Fig 9. Comparing Changes in main and orderHistory

In order to merge changes in, we can create a pull request in GitHub.

However, sometimes when we try to merge changes into the main branch, we encounter conflicts. Here we can see the conflicts found in the orderHistory branch.

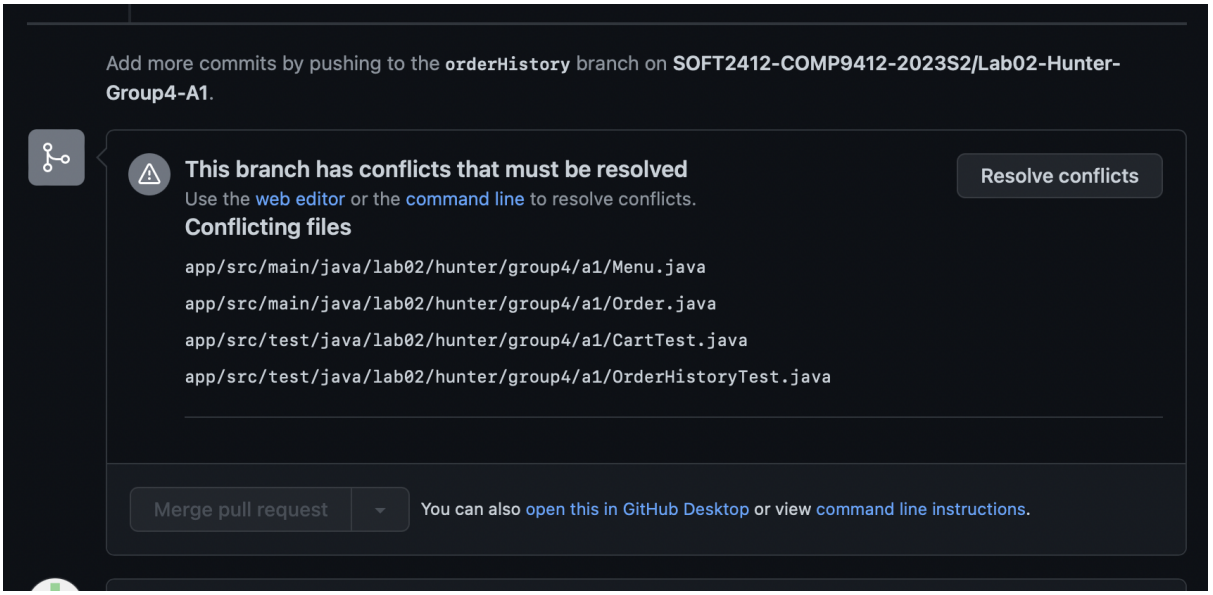


Fig 10. Conflicts

When we try to resolve conflict, for example in Menu.java, we will see:

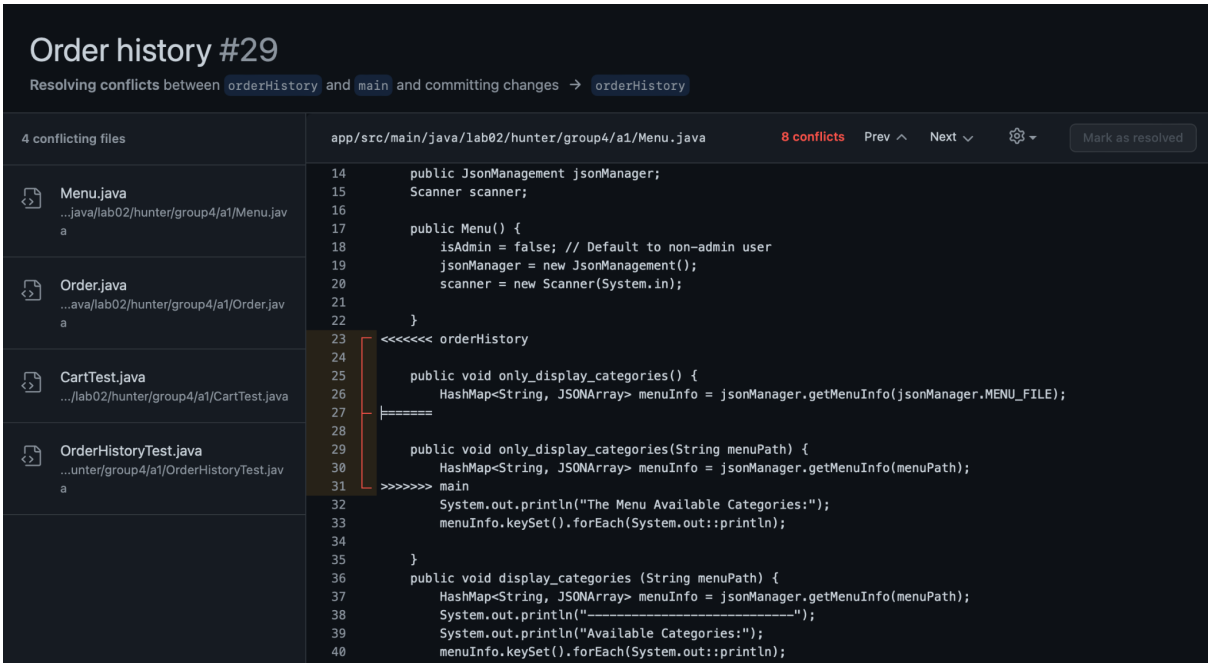


Fig 11. Conflicts in Menu.java

The red lines represent the conflict found. This means that there are two different versions of a similar part of the code.

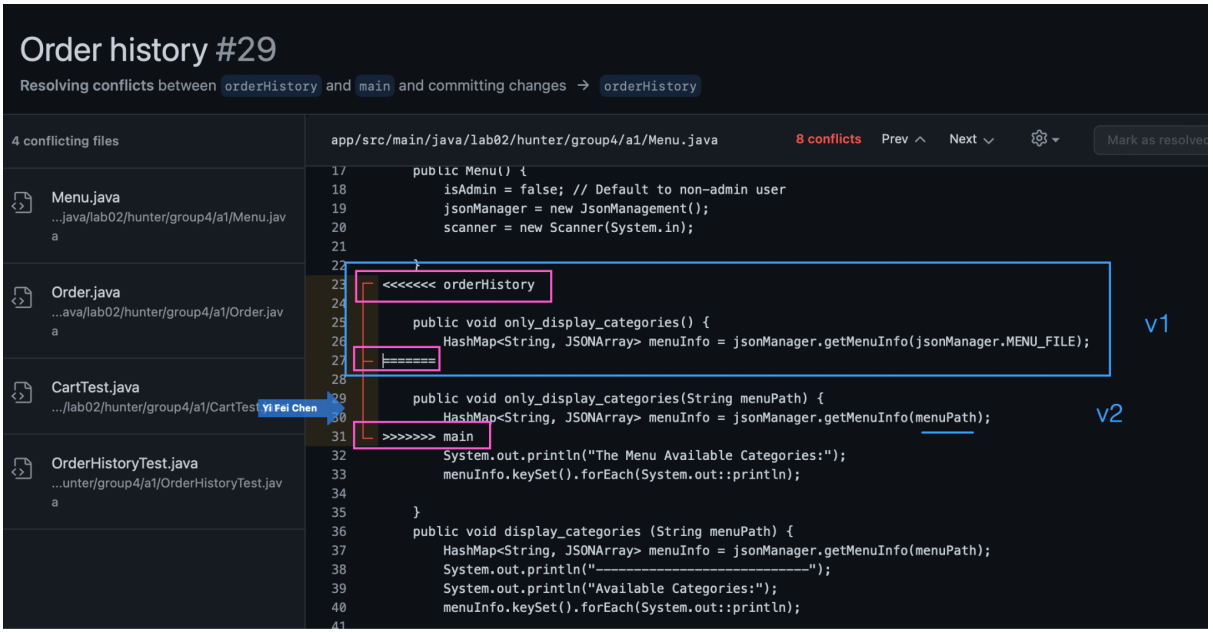


Fig 12. Headers

The pink boxes are the headers. This basically means that the v1 is the version found in orderHistory branch while v2 is the version found from main. Although the two versions look similar, there is a difference in the getMenuInfo().

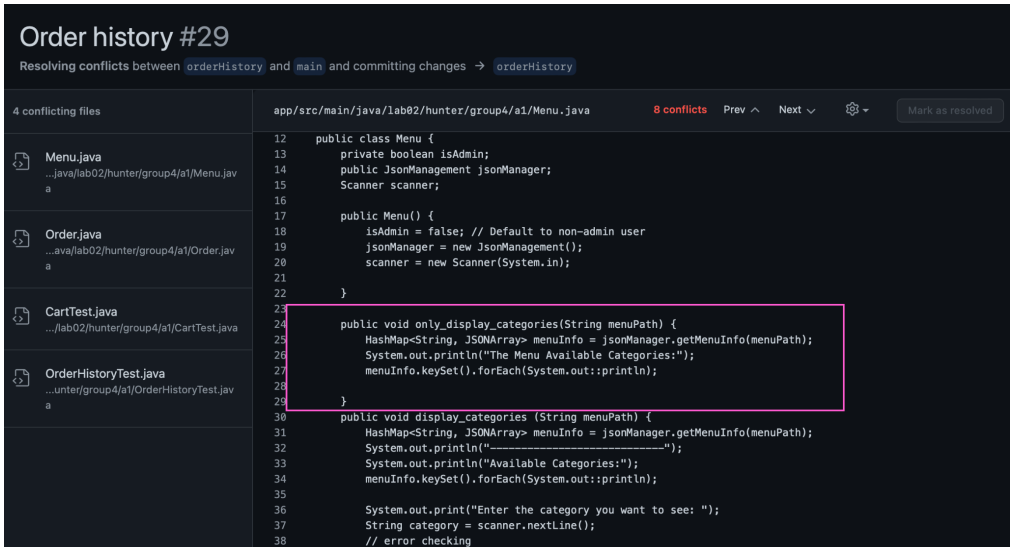


Fig 13. Editing Conflict

We can pick the version we want to use and delete the unused version. In this case, we picked v2, the version from main, and deleted v1.

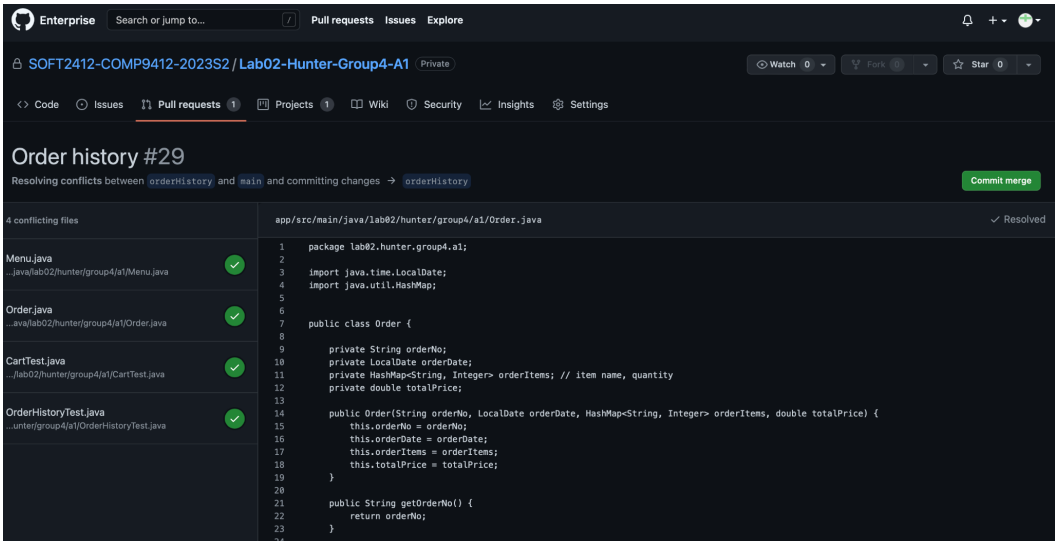


Fig 14. Resolving Conflict

After resolving all the conflicts, we can now commit merge.

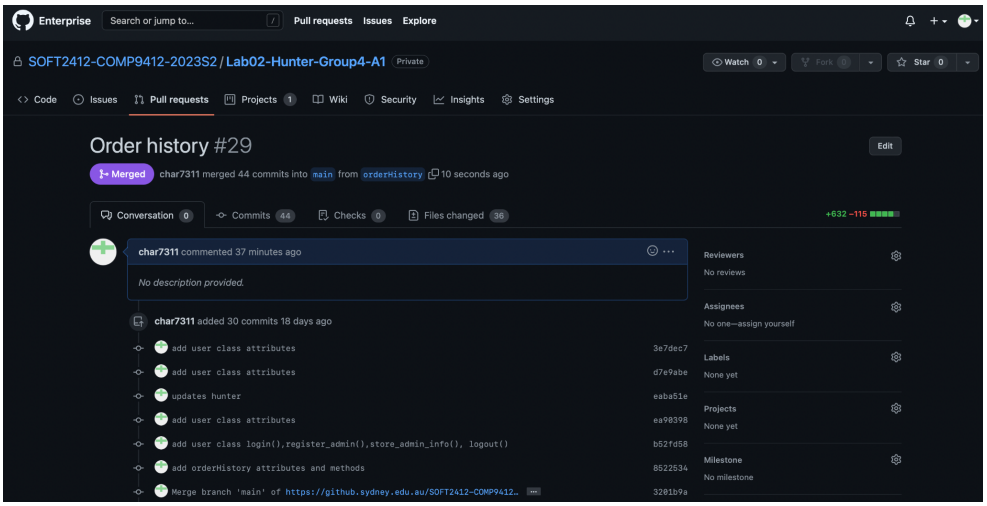


Fig 15. Commit Merge Successfully

3.6 GitHub Collaboration

3.6.1 Feature Branching

In order for everyone to collaborate on GitHub, we used feature branching. So for every feature in the program, we would create a special branch. This means groups members would mostly work on their responsible feature's branch. For example, the group member in charge of user features will work in the Users branch.

We chose this strategy as it allows different members to work on their features simultaneously without affecting each other. Feature branching also means that the main branch would remain unaffected until a feature is fully tested and ready for integration. This would reduce any bugs or issues from happening in main.

3.6.2 Project Boards

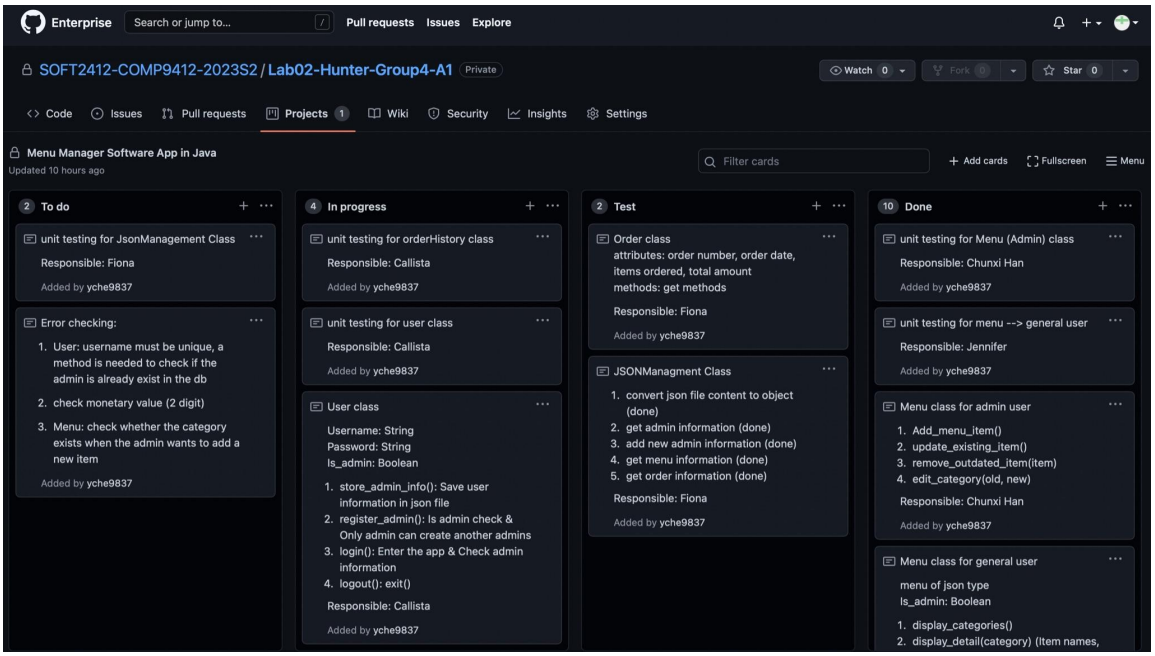


Fig 16. Project Board

For this project, we used GitHub's project board feature to track our project's tasks and monitor our progress. The project board allows us to categorise and visualise our work, distinguishing what needs to be done, what is in progress, which parts are in the testing phase and which part is done. This feature enables us to stay organised and visualise our progress

3.6.3 Benefits of GitHub

GitHub has been proven to be an indispensable tool for our project as its features have allowed us to significantly improve our project development progress. Through GitHub, we were able to maintain a well organised workspace.

GitHub's collaborative feature allowed us to efficiently work together on different aspects of the project without affecting each other's work. Having the pull requests feature also helped in ensuring only approved changes are merged.

4. Gradle

4.1 Objective - How and Why Gradle is Used

Gradle is used for the Menu Application Project as a build automation tool and dependency management system. Here's how Gradle is employed and why:

1. Dependency Management: Gradle is used to manage project dependencies. In the “build.gradle” file, dependencies on external libraries such as JSON parsing libraries (json-simple (version: 1.1.1) and Jackson (version: 2.12.3)), testing frameworks (JUnit Jupiter), and other utilities (Mockito and Guava) are defined. Gradle automatically downloads and manages these dependencies, ensuring that the project has access to the required libraries.

2. Compilation: Gradle compiles the Java source code in the project. It automatically detects the source files and compiles them into bytecode, producing “.class” files. These compiled classes are stored in the “build/classes” directory.

3. Testing: Gradle is used for running tests. The “./gradlew test” command executes unit tests defined in the project using testing frameworks like JUnit Jupiter and Mockito. Test results are displayed in the console, indicating whether tests have passed or failed.

4. Custom Tasks: Gradle allows the definition of custom tasks. In the Menu Project, a custom task called “runInteractive” is defined in build.gradle. This task is used to run the application interactively from the command line, facilitating user interaction for testing and demonstration.

5. Code Coverage Reporting: Gradle, in conjunction with the Jacoco plugin, is used to generate code coverage reports. The “./gradlew jacocoTestReport” command generates HTML and CSV reports that provide insights into how much of the codebase is covered by unit tests.

4.2 Why Gradle is Used for the Menu Project

1. Dependency Management: Gradle simplifies the management of project dependencies. It automatically resolves and downloads dependencies from repositories like Maven Central, reducing the manual effort required for dependency management.

2. Build Automation: Gradle automates the build process. Developers can execute tasks like compilation, testing, and code coverage reporting with simple Gradle commands (./gradlew). This automation saves time and ensures consistency in the build process.

3. Customization: Gradle allows for extensive customization. The `build.gradle` file can be tailored to project-specific requirements, including custom tasks, plugins, and configurations.

4. Compatibility: Gradle is compatible with various IDEs (Integrated Development Environments) and continuous integration tools, making it suitable for collaborative development and integration into development pipelines.

5. Community and Ecosystem: Gradle has a large and active community, which means access to a wealth of resources, plugins, and support. This makes it a reliable choice for building and managing Java projects.

Therefore, Gradle is used for the Menu Application Project to streamline the build process, manage dependencies, run tests, and generate code coverage reports. Its flexibility and automation capabilities make it an essential tool for Java application development, ensuring that the project is well-structured, efficient, and maintainable.

4.3 Additional Tasks/Dependencies

In this Menu Application project, our team used several tasks and dependencies beyond the basic setup. Here's the instruction of the extra tasks and dependencies:

Extra Tasks:

1. Custom Task for Running the Program Interactively:
   - We defined a custom Gradle task named “runInteractive” to run the program interactively. This task allows users to use the command line: “./gradlew runInteractive --console=plain” to easily demonstrate and test the Menu Application by providing input interactively through the console.

```
63    // custom task for running the program interactively using gradle
64    task runInteractive(type: JavaExec) { // for running the program interactively
65    // i.e. demo using ./gradlew runInteractive --console=plain
66        main = 'lab02.hunter.group4.a1.App'
67        classpath = sourceSets.main.runtimeClasspath
68        standardInput = System.in
69    }
```

Fig 17.

Extra Dependencies:

1. JaCoCo (Java Code Coverage Library):
   - The JaCoCo plugin and its dependencies can generate code coverage reports for our menu project.
   - JaCoCo helped us measure code coverage, providing insights into how well the tests covered our codebase.

```
plugins {
    // Apply the application plugin to add support for building a CLI application in Java.
    id 'application'
    id 'jacoco'
}
```

Fig 18.

2. JUnit Jupiter and Mockito:
   - The Gradle includes dependencies for testing frameworks, JUnit Jupiter and Mockito.
   - JUnit Jupiter is able to write and run tests for your application, ensuring its correctness.
   - Mockito facilitated the creation of mock objects for testing, making it easier to isolate and test specific components of the menu application.

```
dependencies {
    // Use JUnit Jupiter for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'
    testImplementation 'org.mockito:mockito-core:3.12.4'
```

Fig 19.

3. Guava (Google Guava Library):
   - The Gradle is included the Guava library as a project dependency.
   - Guava provides various utility classes and methods that can simplify tasks related to collections, caching, and more.
   - While Guava wasn't directly used in the provided code snippets, it can be handy for more advanced functionality in our application.

```
dependencies {
    // Use JUnit Jupiter for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'
    testImplementation 'org.mockito:mockito-core:3.12.4'

    // This dependency is used by the application.
    implementation 'com.google.guava:guava:31.0.1-jre'
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.0'
    implementation 'com.googlecode.json-simple:json-simple:1.1.1'
}
```

Fig 20.

4. Jackson and JSON Simple:
   - The Gradle includes dependencies for JSON handling libraries, Jackson and JSON Simple.
   - Jackson and JSON Simple is able to work with JSON data, which is crucial for reading and writing JSON files for the Menu Application.
   - These libraries facilitated the parsing and serialisation of JSON data, making it easier to manage menu items, orders, and admin information in the menu application.

```
dependencies {
    // Use JUnit Jupiter for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'
    testImplementation 'org.mockito:mockito-core:3.12.4'

    // This dependency is used by the application.
    implementation 'com.google.guava:guava:31.0.1-jre'
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.0'
    implementation 'com.googlecode.json-simple:json-simple:1.1.1'
}
```

Fig 21.

4.4 Brief Explanation of the build.gradle File

The build.gradle file is well-structured for the menu application, including dependencies for JUnit Jupiter, Mockito, and other required libraries. It also sets up Jacoco for code coverage reporting and defines a custom task for running the program interactively.

```
/*
 * This file was generated by the Gradle 'init' task.
 *
 * This generated file contains a sample Java application project to get you started.
 * For more details take a look at the 'Building Java & JVM projects' chapter in the Gradle
 * User Manual available at https://docs.gradle.org/7.5.1/userguide/building_java_projects.html
 */

plugins {
    // Apply the application plugin to add support for building a CLI application in Java.
    id 'application'
    id 'jacoco'
}

repositories {
    // Use Maven Central for resolving dependencies.
    mavenCentral()
}

dependencies {
    // Use JUnit Jupiter for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'
    testImplementation 'org.mockito:mockito-core:3.12.4'

    // This dependency is used by the application.
    implementation 'com.google.guava:guava:31.0.1-jre'
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.0'
    implementation 'com.googlecode.json-simple:json-simple:1.1.1'
}

application {
    // Define the main class for the application.
    mainClass = 'lab02.hunter.group4.a1.App'
}

jacocoTestReport {
    reports {
        xml.required = false
        csv.required = false
    }
}

test {
    useJUnitPlatform()
    test.finalizedBy jacocoTestReport
}

tasks.named('test') { Task it ->
    // Use JUnit Platform for unit tests.
    useJUnitPlatform()
}
```

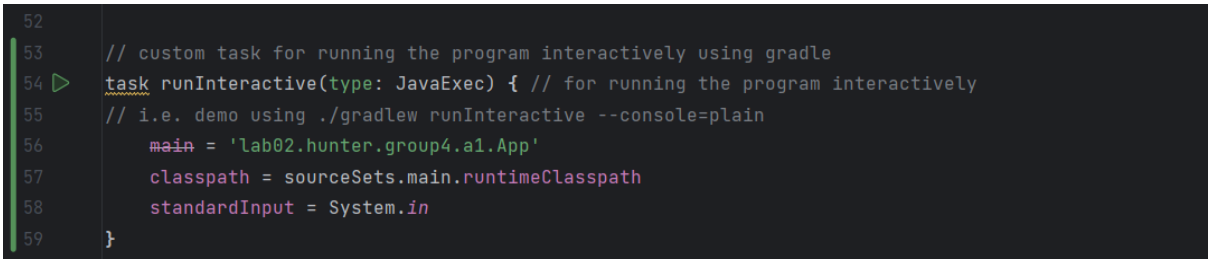


Fig 22.

1. Plugins Section:

- application: This plugin is applied to support building a CLI (Command Line Interface) application in Java. It helps define the main class that should be executed when the application is run.
- jacoco: This plugin is used to generate code coverage reports.

2. Repositories Section:

The project specifies the Maven Central repository for resolving dependencies. Maven Central is a popular repository for Java libraries and dependencies.

3. Dependencies Section:

- testImplementation: Dependencies required for testing the application.

  org.junit.jupiter:junit-jupiter:5.8.2: The JUnit Jupiter framework for unit testing.

  org.mockito:mockito-core:3.12.4: The Mockito library for mocking objects in tests.

- implementation: Dependencies required for the application to run.

  com.google.guava:guava:31.0.1-jre: Google Guava, a widely used library that provides utilities for common programming tasks.

  com.fasterxml.jackson.core:jackson-databind:2.13.0: The Jackson library for working with JSON data.

  com.googlecode.json-simple:json-simple:1.1.1: JSON Simple, a lightweight library for working with JSON data.

4. Application Section:

mainClass: Specifies the fully-qualified name of the main class for the application. In this case, it points to the App class in the lab02.hunter.group4.a1 package.

5. Jacoco Test Report Section:

- jacocoTestReport: Configuration for generating code coverage reports.
- *“xml.required”* and “*csv.required”* are set to false, indicating that XML and CSV reports are not required for this project.

6. Test Section:

useJUnitPlatform(): Configures the use of the JUnit Platform for running unit tests.

“test.finalizedBy jacocoTestReport”: Specifies that the jacocoTestReport task should be executed after the tests are run.

7. Custom Task Section (runInteractive):

Defines a custom Gradle task called runInteractive. This task is used for running the program interactively from the command line. It specifies the main class (lab02.hunter.group4.a1.App) to execute, sets the classpath, and configures standard input to read from the console.

This build.gradle script automates various aspects of the project, including dependency management, testing, and code coverage reporting. It simplifies the build process and makes it easier to manage project dependencies and tasks. Additionally, it provides a custom task for interactive program execution during development and testing.

To run the program interactively using Gradle, user can use the following command in the terminal:

./gradlew runInteractive --console=plain

This command will execute the main class (lab02.hunter.group4.a1.App) and allow you to interact with the program through the console.

Make sure users have the required Gradle wrapper (gradlew) in the project directory to run these tasks.

4.5 How did those tasks/dependencies helped to build application?

The tasks and dependencies defined in the build.gradle file play a crucial role in building the Menu Application. The report will explain how each of them contributes to the development and build process:

1. Dependencies Section:
Dependencies are specified in the “dependencies” section. These are external libraries and tools that the application relies on.

- For example, the application uses JUnit Jupiter (org.junit.jupiter:junit-jupiter) for testing and Mockito (org.mockito:mockito-core) for mocking objects during testing.
- Guava (com.google.guava:guava) provides utilities that simplify various programming tasks.
- Jackson (com.fasterxml.jackson.core:jackson-databind) is used for working with JSON data, which is important for reading and writing menu and order information.
- JSON Simple (com.googlecode.json-simple:json-simple) is used for handling JSON data in a lightweight manner.
- These dependencies are automatically downloaded and managed by Gradle, making it easy to include external functionality in the application.

2. Application Plugin:

- The *id 'application'* plugin is applied to enable the creation of a CLI (Command Line Interface) application.
- It allows you to define the main class to be executed when running the application. In this case, it points to the App class in the “lab02.hunter.group4.a1” package.
- This simplifies the process of running the application from the command line.

3. Jacoco Plugin and Test Section:

- The *id 'jacoco'* plugin is used to generate code coverage reports, which help assess the quality of the tests.
- The *'test'* task is configured to use the JUnit Platform for running unit tests.
- After the tests are executed, the *'jacocoTestReport'* task generates code coverage reports.
- These reports provide insights into which parts of the code are covered by tests and which parts may require additional testing.

4. Custom Task (runInteractive):
The custom Gradle task “runInteractive” is defined to run the program interactively.

- It specifies the main class to execute and configures the classpath.
- By using “standardInput = System.in”, it allows the program to read input from the console interactively.

- This task facilitates the interactive testing and demonstration of the Menu Application by running it directly from the Gradle build.

These elements automate various aspects of the build and development process, making it more efficient and ensuring the reliability and quality of the Menu Application.

4.6 Relevant Gradle commands used

The team used various Gradle commands to automate the build process of the Menu Application:

- ./gradlew build: This command compiles the source code, runs tests, and packages the application.
- ./gradlew clean test: Used to execute unit tests defined in the project.
- ./gradlew jacocoTestReport: Generates code coverage reports using Jacoco.
- ./gradlew runInteractive --console=plain: A custom task for running the program interactively.

4.7 Explanation about the results/outputs obtained from relevant Gradle commands

1 ./gradlew build: This command compiles the code and produces the following outputs:

- Compiled .class files in the build/classes directory.
- A JAR file containing compiled classes and resources in the build/libs directory.

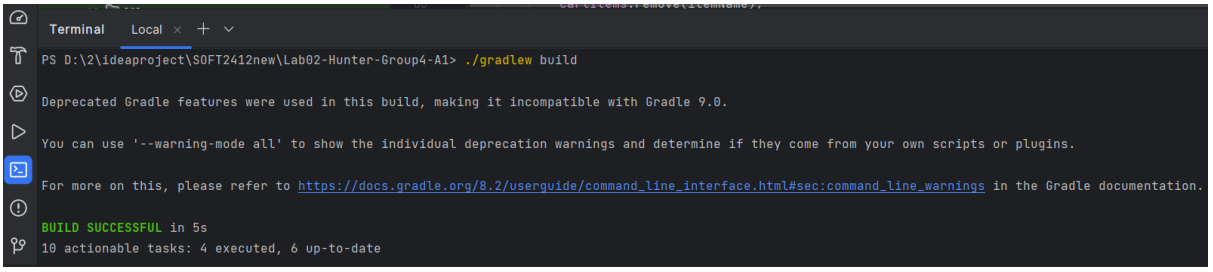


Fig 23. ./gradlew build

2 ./gradlew clean test: Executes unit tests and produces test results. It will display test success or failure status in the console.

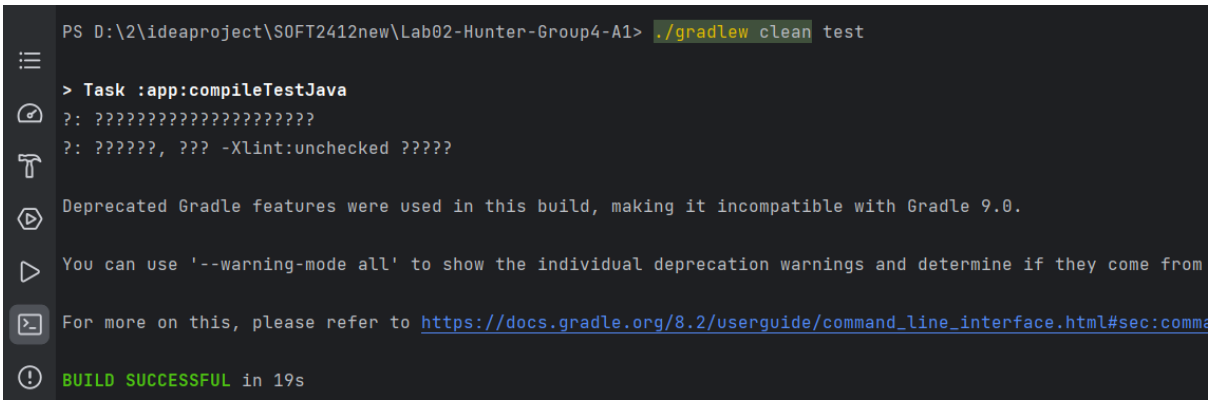


Fig 24. ./gradlew clean test

3 ./gradlew jacocoTestReport: Generates code coverage reports in HTML and CSV formats. The HTML report is available in the build/reports/jacoco/test/html directory.

Fig 25. ./gradlew jacocoTestReport

4 ./gradlew runInteractive --console=plain: This custom task allows running the application interactively, providing a console interface for user interaction.

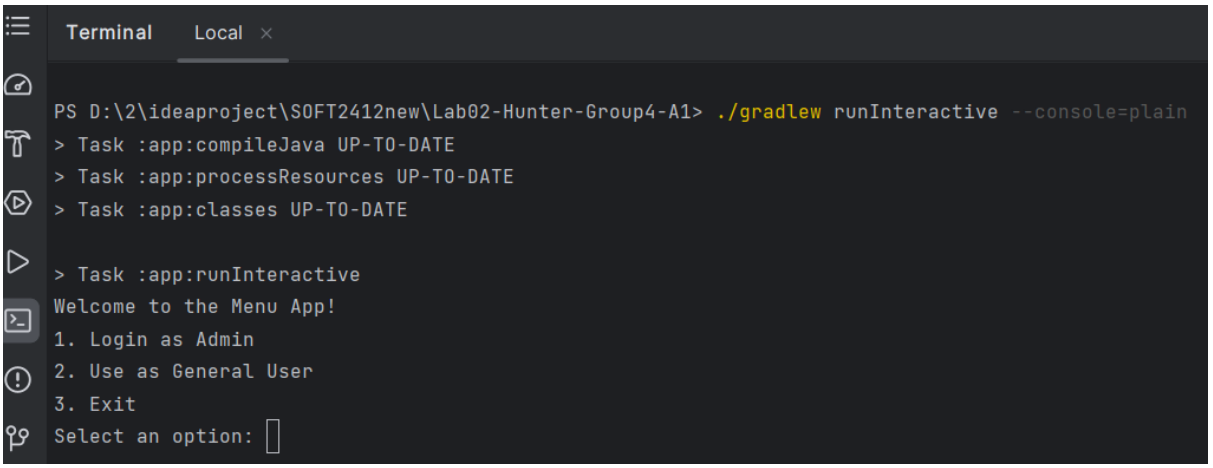


Fig 26. ./gradlew runInteractive --console-plain

4.8 The Produced Artefacts

- Compiled Classes: The build/classes directory contains compiled Java classes.

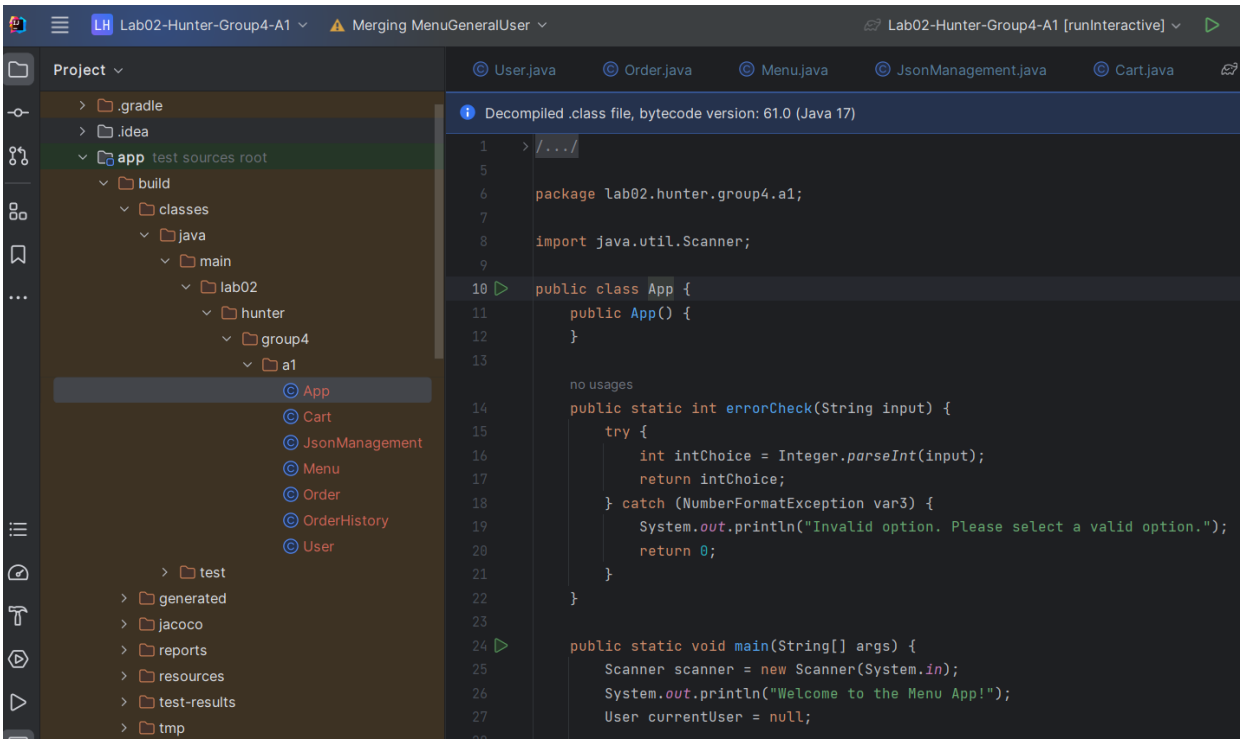


Fig 27. Compiled Classes

- JAR File: The build/libs directory contains a JAR file that can be used to run the application.

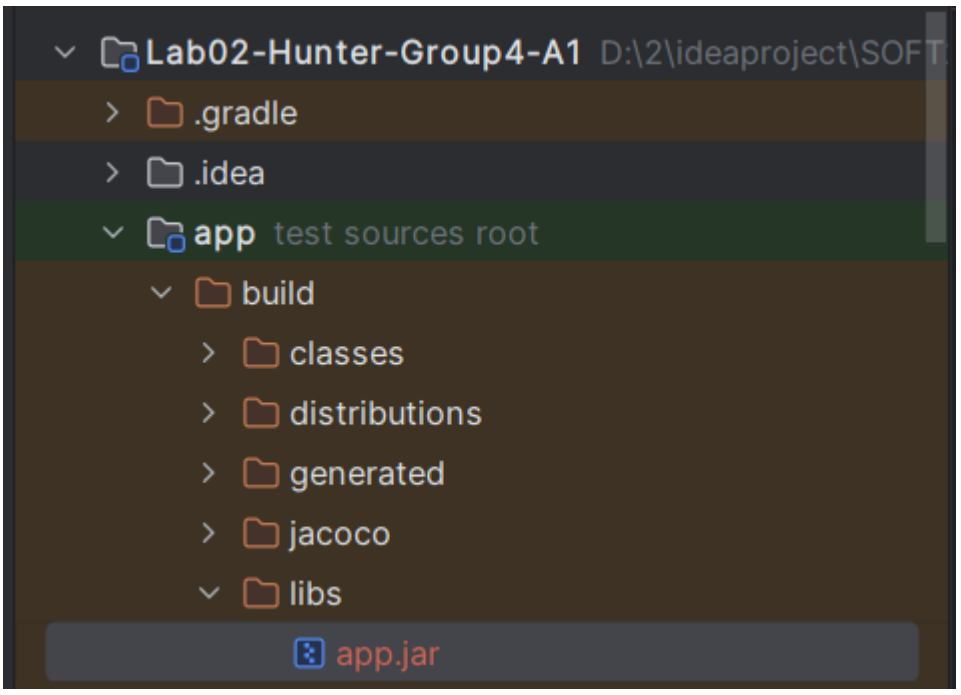


Fig 28. JAR file

- Code Coverage Reports: The build/reports/jacoco/test/html directory contains HTML code coverage reports. These reports show which parts of the code are covered by tests and which are not.

app > lab02.hunter.group4.a1

lab02.hunter.group4.a1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
App		0%		0%	22	22	96	96	3	3	1	1
Cart		82%		62%	11	26	28	103	1	8	0	1
Menu		93%		69%	13	34	16	148	0	10	0	1
User		87%		72%	6	17	7	39	1	8	0	1
OrderHistory		86%		75%	4	11	4	27	1	5	0	1
JsonManagement		98%		100%	0	14	4	81	0	9	0	1
Order		100%		100%	0	7	0	17	0	6	0	1
Total	389 of 1,835	78%	62 of 145	57%	56	131	155	511	6	49	1	7

Fig 29. Code Coverage Reports

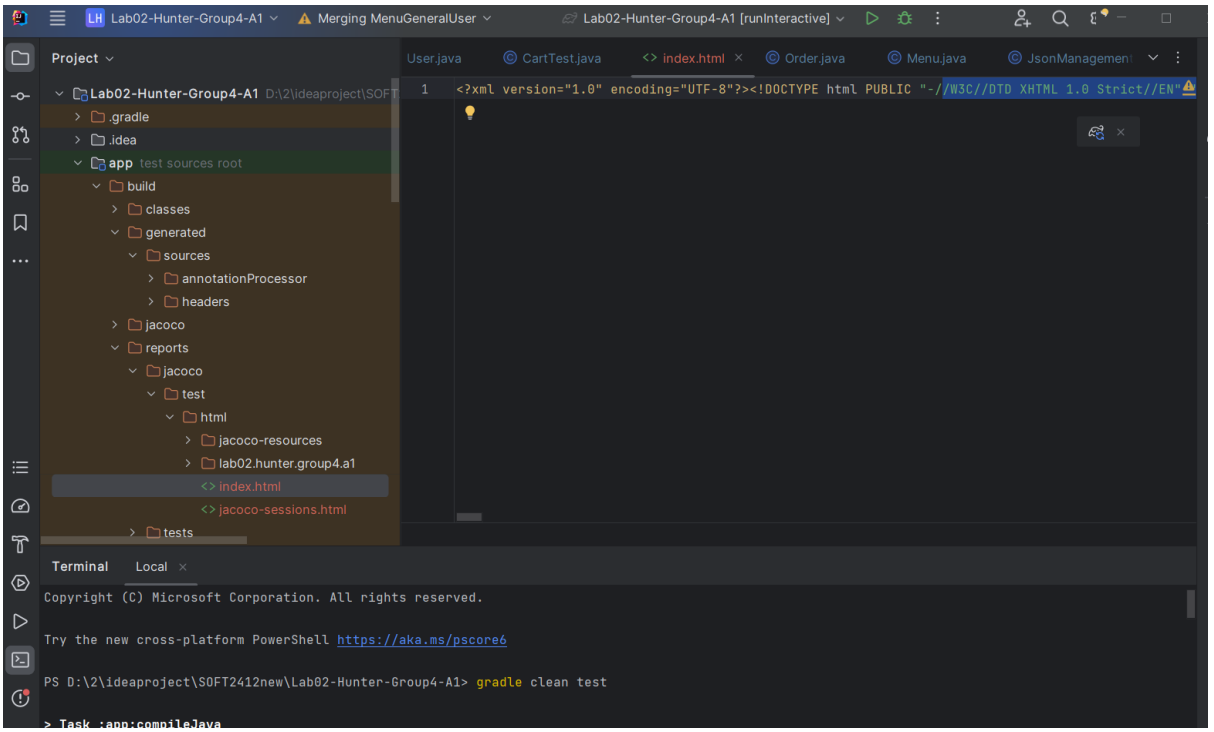


Fig 30. Report index.html

- Test Reports: Test results are displayed in the console when running ./gradlew test.

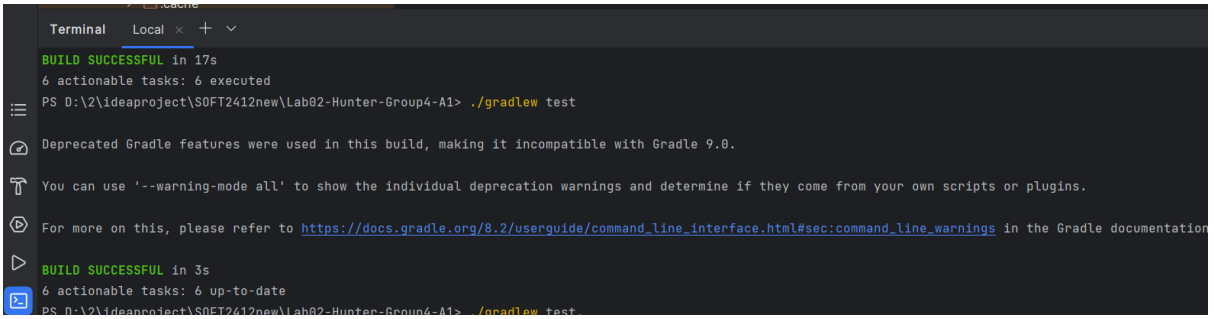


Fig 31

.

- Interactive Application: The ./gradlew runInteractive task allows running the application interactively from the command line.

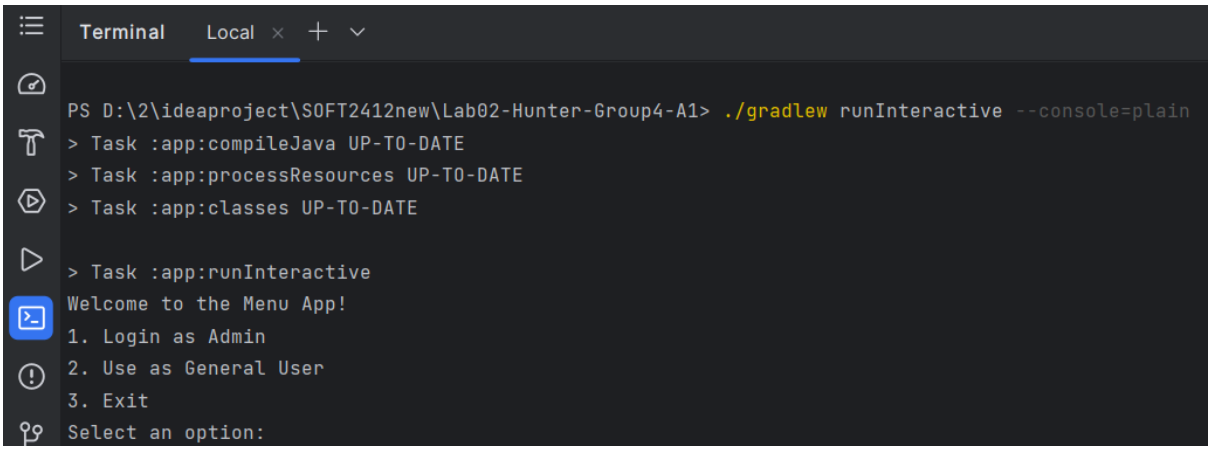


Fig 32.

5. Jenkins

5.1 Objective - How and Why Jenkins is Used

Jenkins is a continuous integration (CI) server which assists in automating the build, testing, and deployment phases of our application development process, specifically through triggering code builds when there is changes made to the source code repository. In terms of this project, our main branch has been used to keep track of the version of the integrated code, whenever alterations has been made to the code on the main branch, Jenkins serves a significant role in automatically triggering a series of named tasks to ensure the application's integrity and readiness for deployment i.e. whether the alterations has result in a successful build or otherwise.

The reason to employ Jenkins in the project can be summarised into three main reasons, namely the rapid feedback from each alterations in the latest version of the application, quality assurance, and streamlining the deployment. From the outset, due to the automatic nature of the triggering of the specified tasks (for e.g. build and clean), the instant information on the impact of the latest code changes in the specified branch (e.g. the main branch) and the cause of the problem in the case if the new alterations result in a build fail allows the team to detect and resolve the problem immediately rather than detecting at a later phase without knowing which prior changes has caused the problem. As such, the early detection and the instant feedback streamline the deployment, ensuring consistency and rapid delivery of new features to our usage and testing stage.

5.2 Automating the CI Pipeline

Our team has invested significant effort into automating the CI pipeline, where the details of the each step will be discussed later in Jenkins part of the report. The typical process in chronological order is as the following:

5.2.1 Setting up Jenkins

Our team has chose to set up jenkins using Docker, a software that allows delivering software in containers. After docker is installed, a jenkins server is run inside a docker container using the command below:

```
docker run -d -v jenkins_home:/var/jenkins_home -p 8080:8080 -p 50000:50000 --name
myjenkins jenkins/jenkins:lts
```

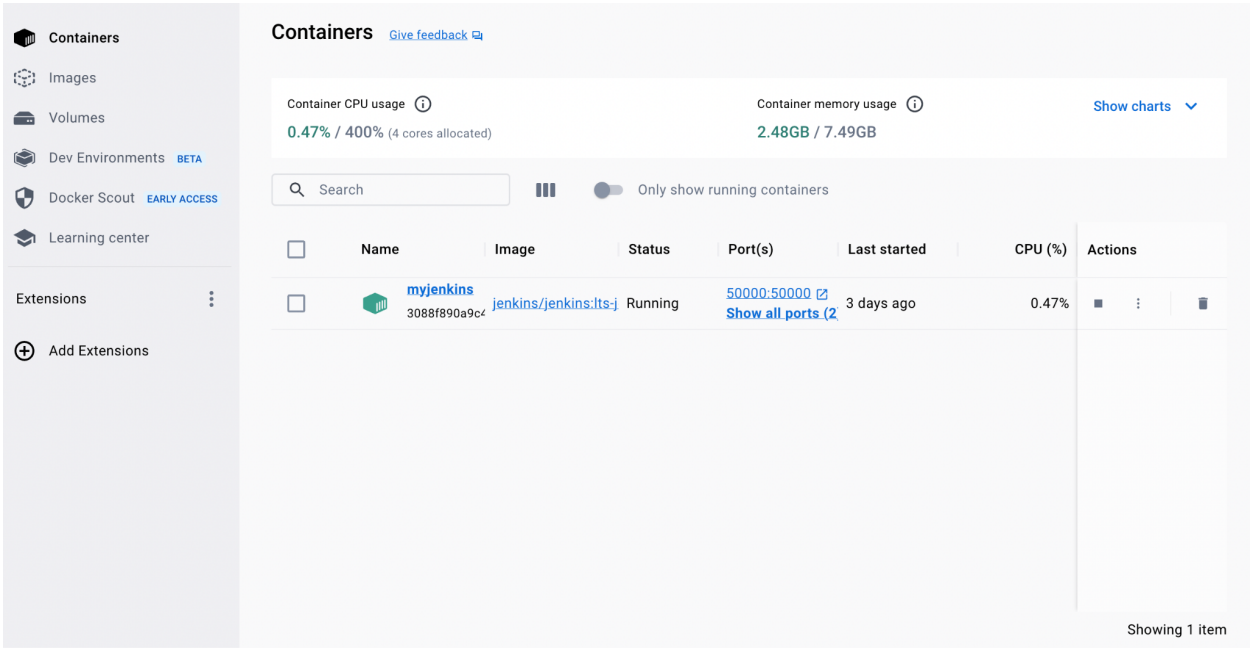


Fig 33. Jenkins server named ‘myjenkins’ running inside a docker container

Note that in the command above, the port 8080 on the host has been mapped to port 8080 on the container. Jenkins by default runs on port 8080, allowing our team to access the Jenkins web interface via my host machine’s IP address on port 8080. Therefore, after the execution of the aforementioned command, our team is able to access the jenkins web interface by navigating to http://localhost:8080 on browser such as chrome.

The setting up of jenkins initiated with the prompt to unlock jenkins via the visit to http://localhost:8080, which could be obtained by executing the following commands:

```
docker exec -it jenkins /bin/bash
cat /var/jenkins_home/secrets/initialAdminPassword
```

After unlocking jenkins, install the suggested plugin and create the admin credentials to proceed, which will be used as future jenkins login information to access the jenkins interface.

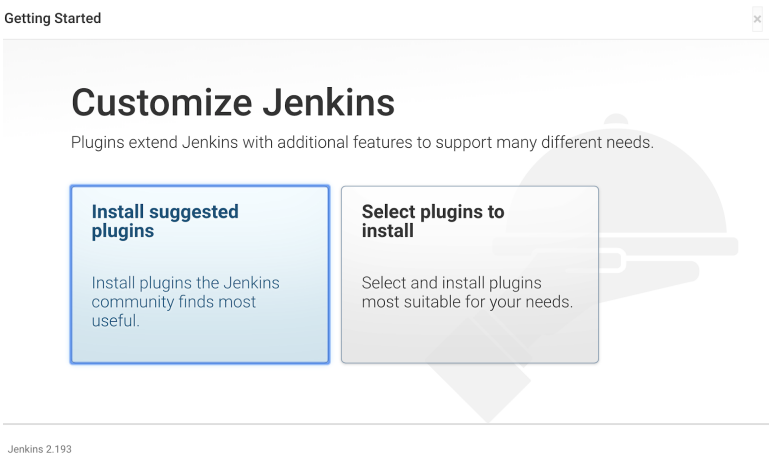


Fig 34. Installing plugins

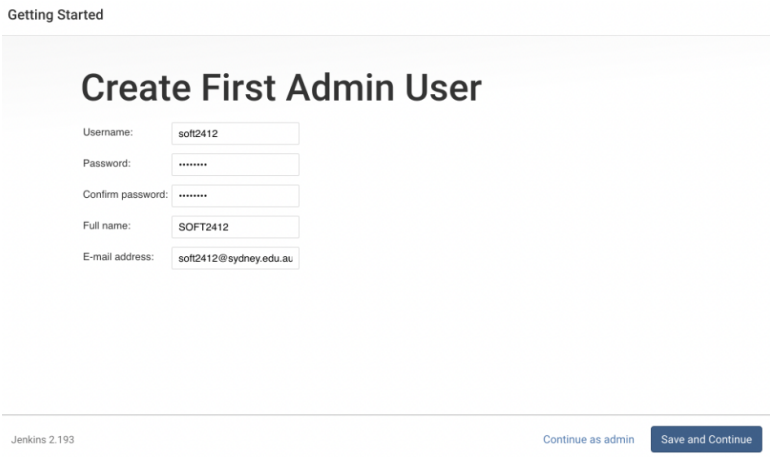


Fig 35. Create admin credentials

Last step of setting up Jenkins is to confirm the instance configuration to local host i.e. http://localhost:8080/

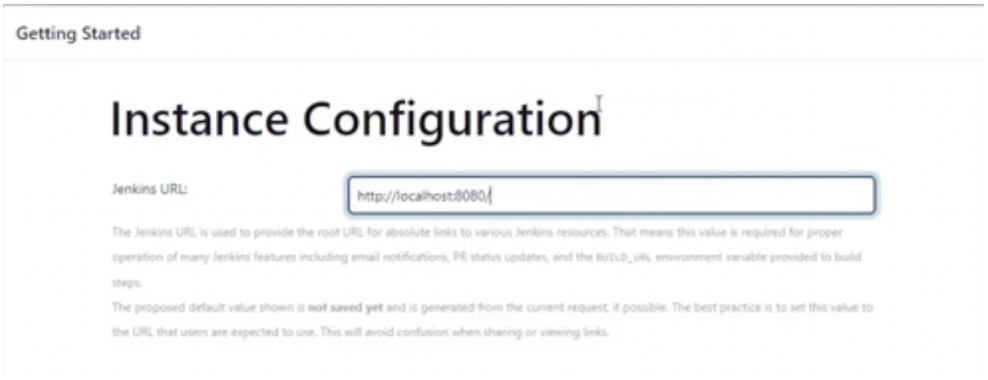


Fig 36. Instance configuration of Jenkins

In the case where admin credential has been forgotten, all the caches including images and volumes of the associated jenkins server needs to be removed all the steps above needs to be reperformed.

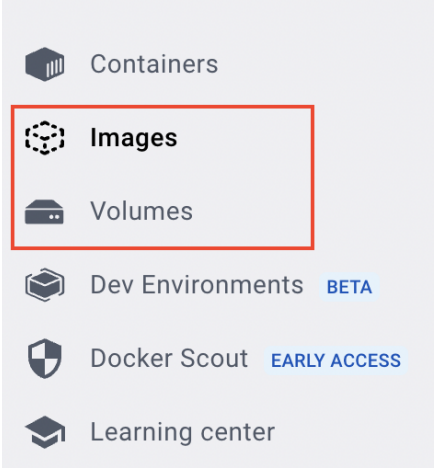


Fig 37. Images and Volumes in docker desktop

5.2.2 Configuration within Jenkins

Before creating the project, we first need to set the Jenkins' API endpoint to usyd enterprise github, specifically https://github.sydney.edu.au/api/v3. Subsequently, create a freestyle project, which our team has named it as "soft2412 asm1".

Fig 38.

Since Jenkins are non-interative by nature, meaning that Jenkin's build environment does not have a console to read input from. Therefore the two tasks we would like to automatically perform are gradle clean and gradle build, and as output we would like to see the jacoco report regarding test coverage. This means that the two plugins we need to install onto Jenkins are gradle and jacoco.

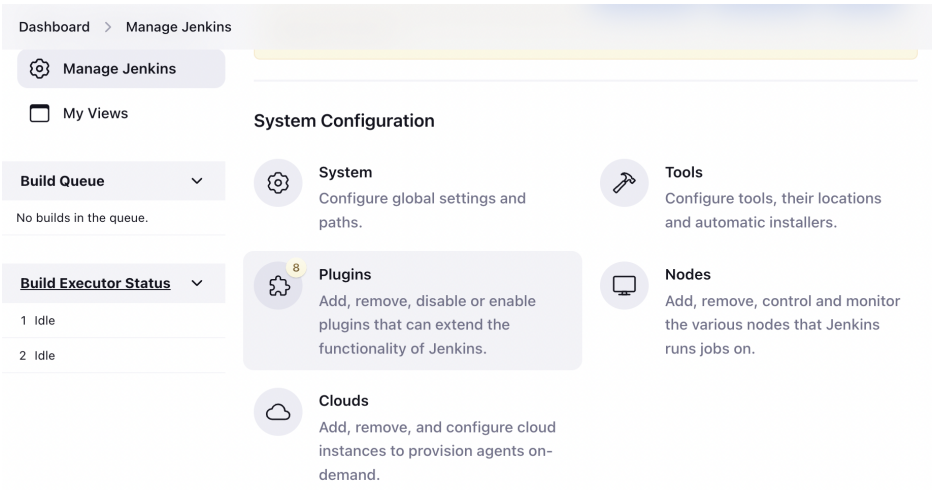


Fig 39. Installing plugins on Jenkins

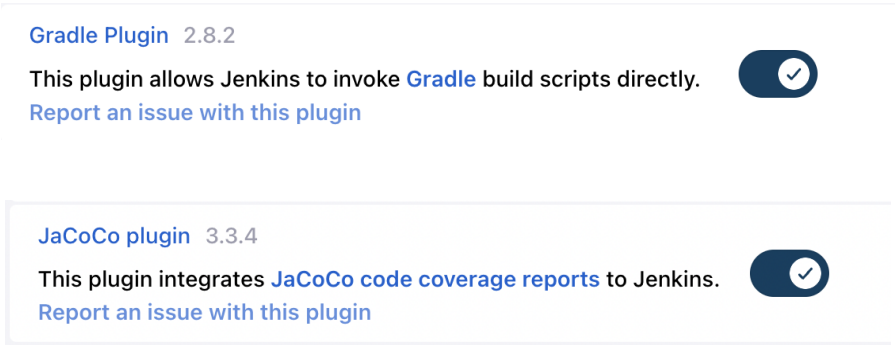


Fig 40. Gradle and JaCoCo plugins in Jenkins interface

5.2.3 GitHub Integration - Configuration and Webhooks

Configuration

With all the needed plugins and dependencies, we need Jenkins to pull the latest code, therefore we have specified the location of the git repository where we stored the code.

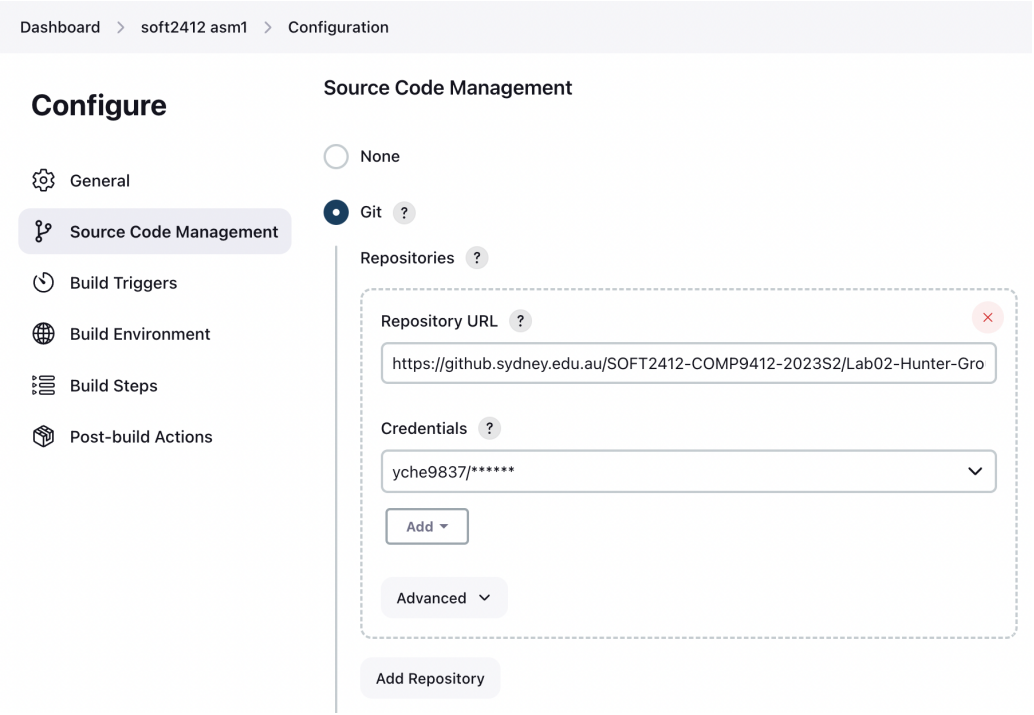


Fig 41. Link Jenkins to GitHub

Last step to integrate Github with Jenkins is to enable the Jenkins project “soft2412 asm1” to listen for webhooks by selecting the GitHub hook trigger for GITScm polling.

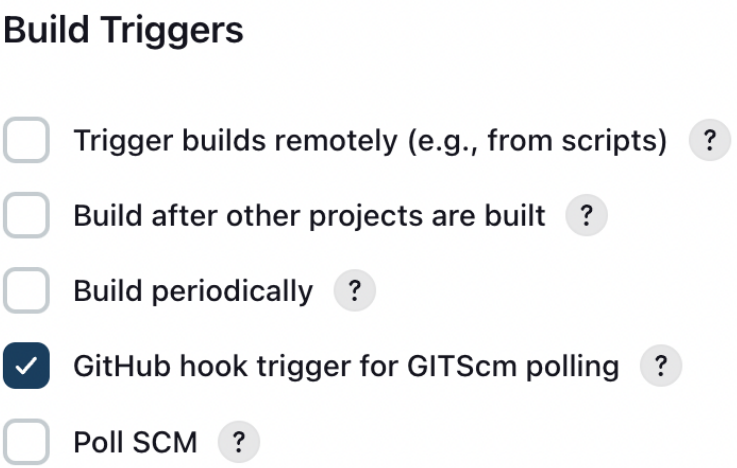


Fig 42. Preparation for webhooks

Webhooks

Integrate Jenkins wih GitHub using web-hooks allows for automated building and testing whenever alterations has been made to the nominated branch (typically the main branch). Specifically, this means that when we want to check if the new code builds or not, instead of the workflow push the code to main → Jenkins web interface → Build Now → Success / Failure, with webhooks, it streamline the process to push the code to main → success/failure.

To implement webhooks, we first exposed Jenkins on the public internet instead of running on localhost at port 8080 using ngrok. By running the command:

```
ngrok http 8080
```

We have the following output which most importantly has provide us with a public IP address, and due to security reason, we are using https: https://3251-2001-8003-29b1-100-901c-68b9-ac4e-e6d2.ngrok-free.app/

```
ngrok                                                                                                    (Ctrl+C to quit)

Introducing Always-On Global Server Load Balancer: https://ngrok.com/r/gslb

Session Status                online
Account                       yche9837 (Plan: Free)
Update                        update available (version 3.3.4, Ctrl-U to update)
Version                       3.0.7
Region                        Australia (au)
Latency                       21ms
Web Interface                 http://127.0.0.1:4040
Forwarding                    https://3251-2001-8003-29b1-100-901c-68b9-ac4e-e6d2.ngrok-free.app -> http://localhost:8080

Connections                   ttl     opn     rt1     rt5     p50     p90
                              60      1       0.03    0.07    0.12    30.10

HTTP Requests
-------------

POST /github-webhook/             200 OK
POST /github-webhook/             200 OK
POST /widget/BuildQueueWidget/ajax 200 OK
```

Fig 43. Output of *ngrok http 8080* on terminal

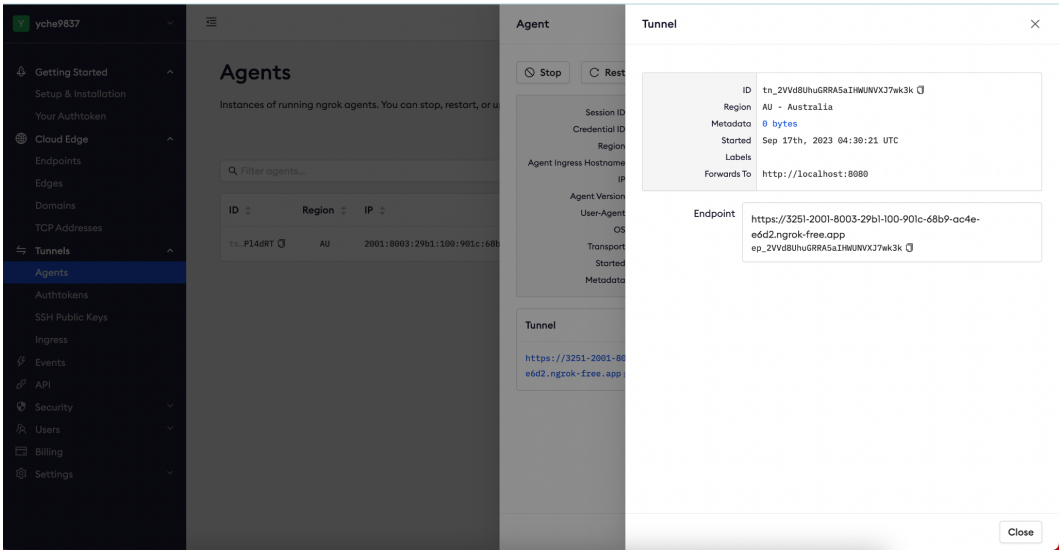


Fig 44. The public IP address associated with local host 8080 on ngrok web interface

Last step to integrate Jenkins and Github is to paste the ngrok ip address to the working repository on github. By adding the webhook with the URL <ngrok ip address> + /github-webhook/ to the “payload URL” input field on github, i.e. https://3251-2001-8003-29b1-100-901c-68b9-ac4e-e6d2.ngrok-free.app/github-webhook/.

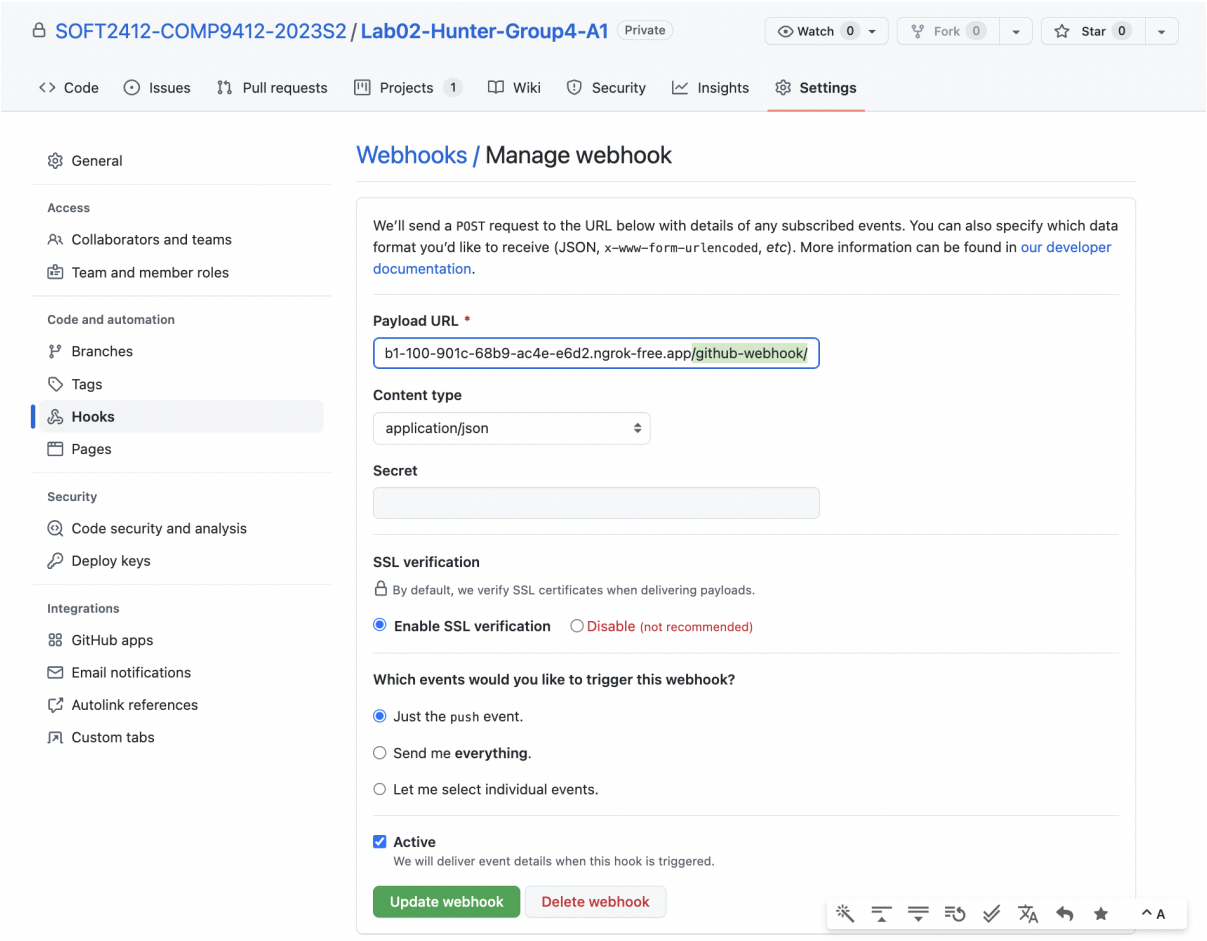


Fig 45. add webhooks on github

If github configures the URL as denoted by a green tick, it means that URL is accessible and GitHub and Jenkins integration is successful and ready to use.

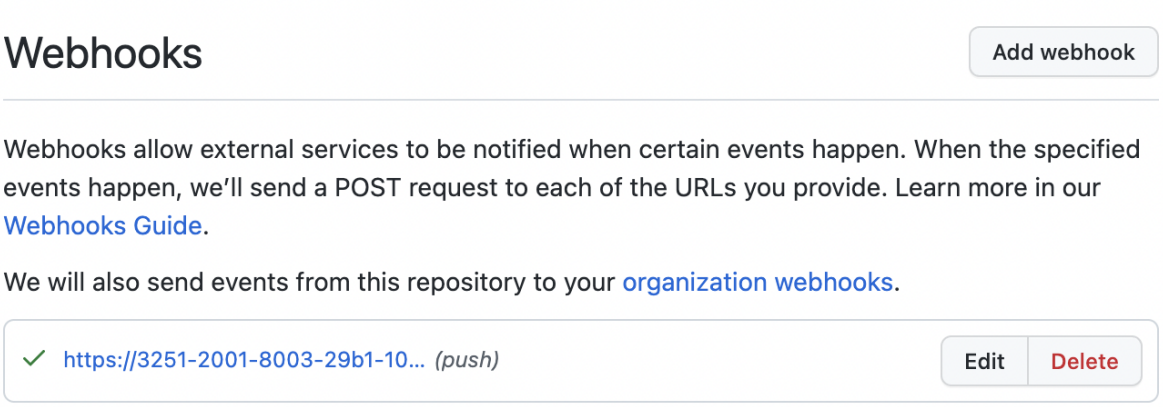
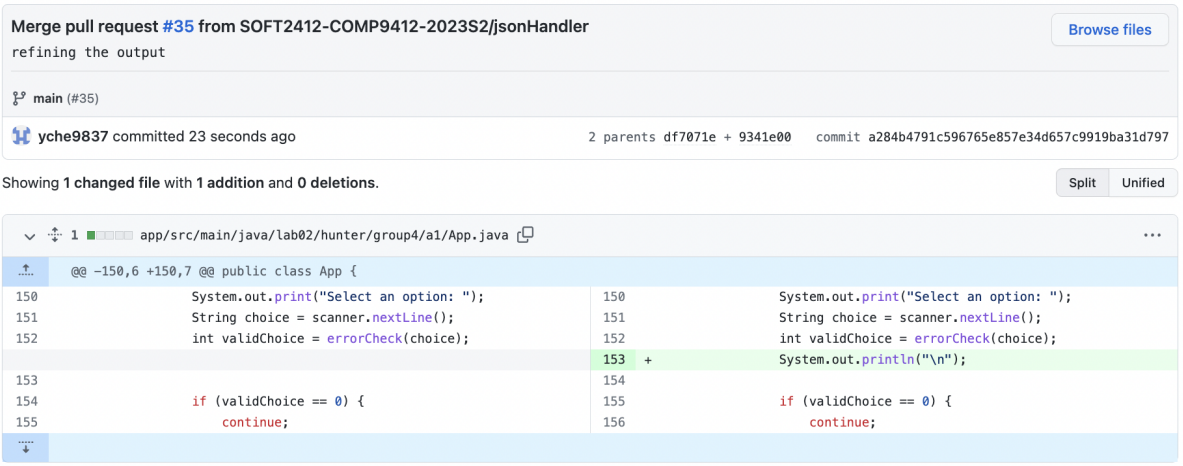


Fig 46. successful adding on webhooks

Figures below shows the automated build after successfully integrating github and jenkins, where a simple change in adding a new line triggers automated build in Jenkins, in this particular case, it builds successfully.



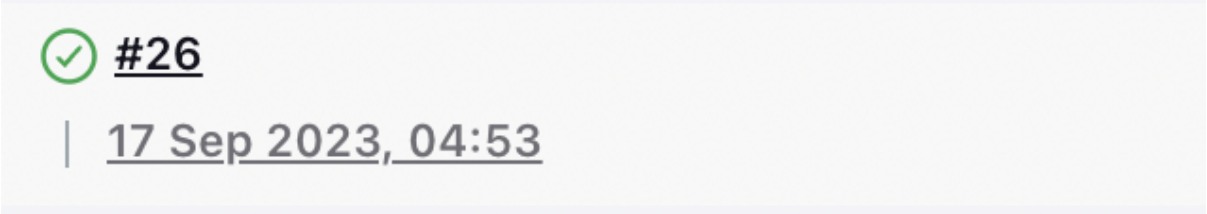


Fig 47.

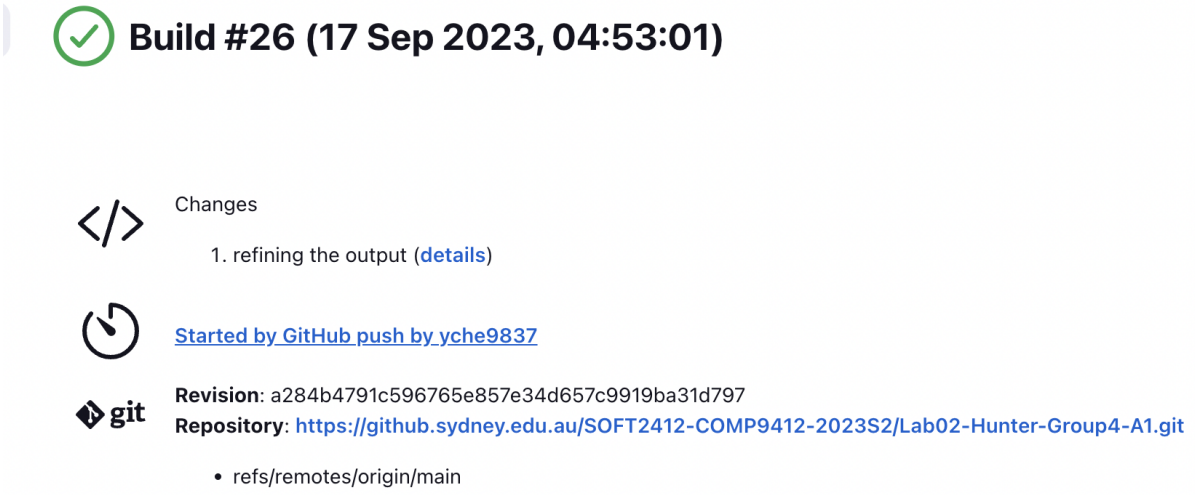


Fig 48. successful integration of Github and Jenkins

5.2.4 Automated Build/Test

For automated build and test, the specific tasks outlined in build.gradle that require Jenkins to run are outlined in the ‘Tasks’ field of the ‘‘Build Steps’ section. The figure below shows how Jenkins will automatically run the clean and build tasks when alterations has occurred in the main branch:

```
Gradle clean

Gradle build
```

Note that the build task also encompasses running all the tests, where results has been stored as .xml files for each individual class tests.

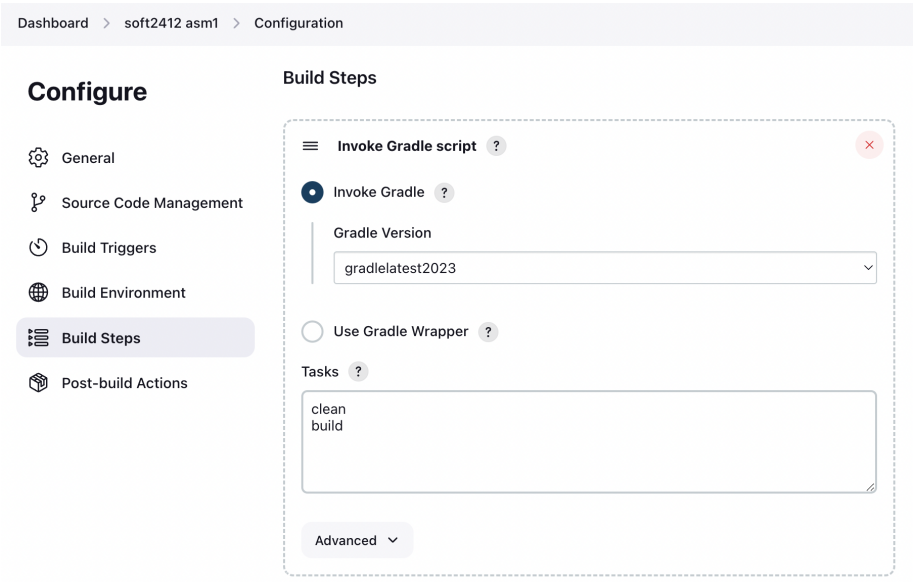


Fig 49. Automated clean and build

A shortcut to produce the JUnit test report rather than entering individual test .xml files is by entering the following into the test report XMLs fields as evident in figure 16.

```
**/test-results/**/*.xml
```

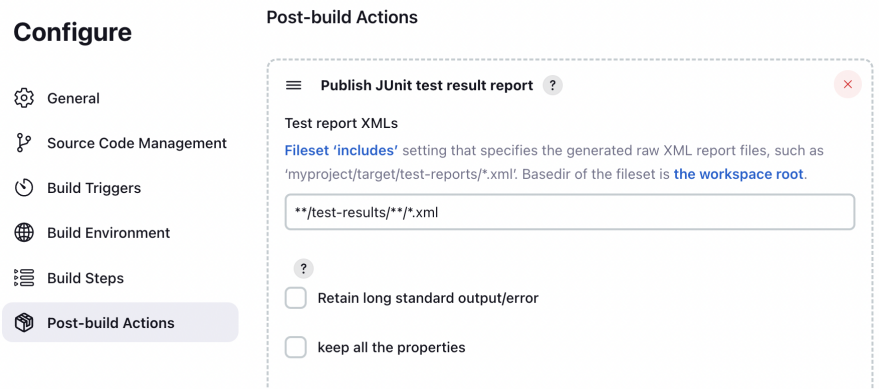


Fig 50. Post-build actions - combine the .XML test to form test reports after each automated build done by Jenkins

5.2.5 JaCoCo coverage report on Jenkins

Similarly, the entered path to the input fields below present a shortcut for Jenkins to perform automated testing on all the existing test classes in order to generate the JaCoCo coverage report on the overall performance of the tests.

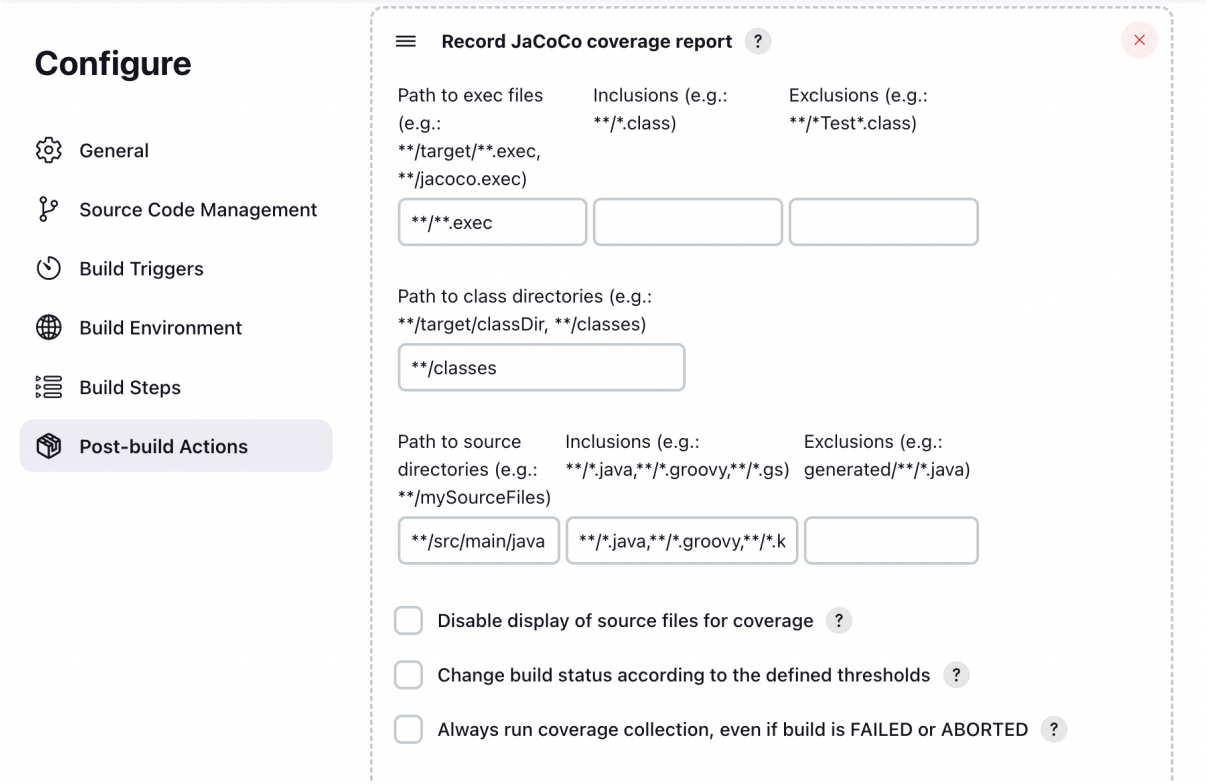


Fig 51. Jenkins configuration for JaCoCo coverage report post build

5.2.6 Explanation on Jenkins Outputs

Below shows an example of the test result from build #21. Here, it contains all the test files associated with the corresponding classes that each tests. Duration refers to how long does each test take to finish running. Each tests within the test files can have three potential outcomes, namely fails, skips and passes. Here, it shows that all tests in the test files have passed.

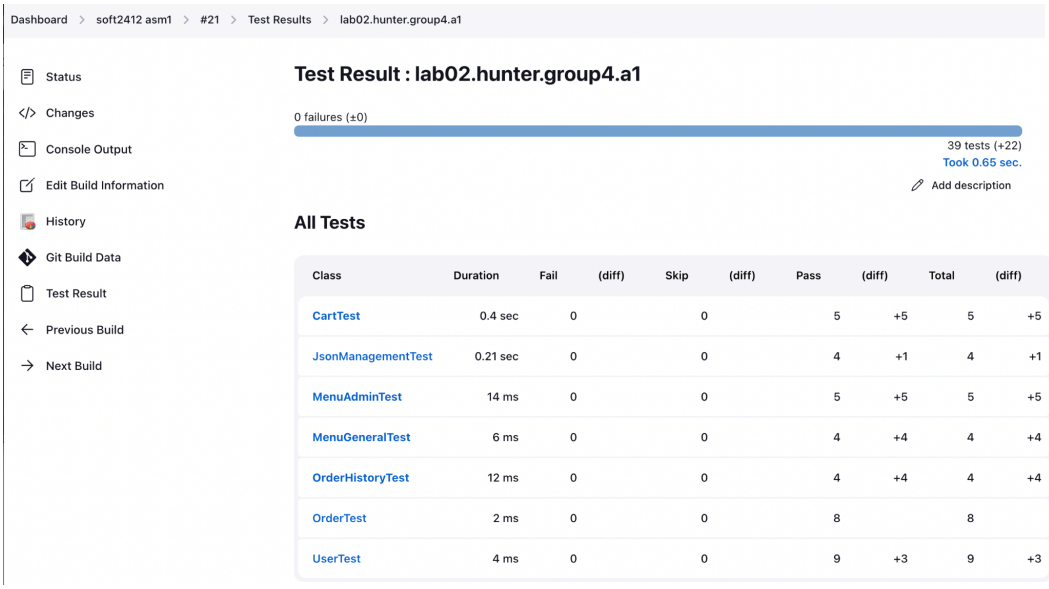


Fig 52. Test Results from Build #21

Another important deliverable of Jenkins is the build log as evident in Figure 19, where green denotes successful builds and red indicates otherwise.

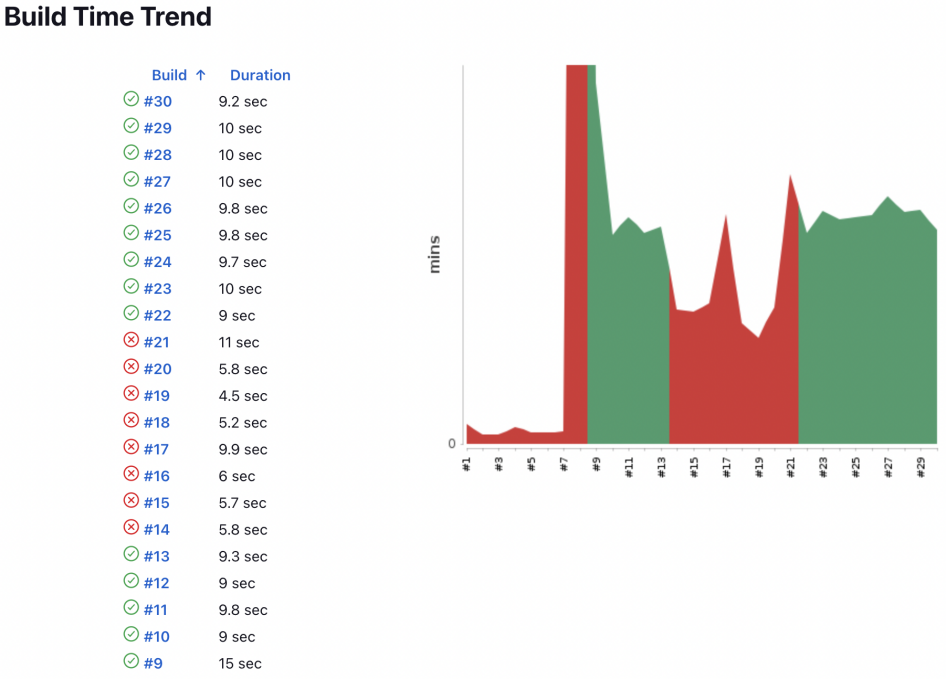
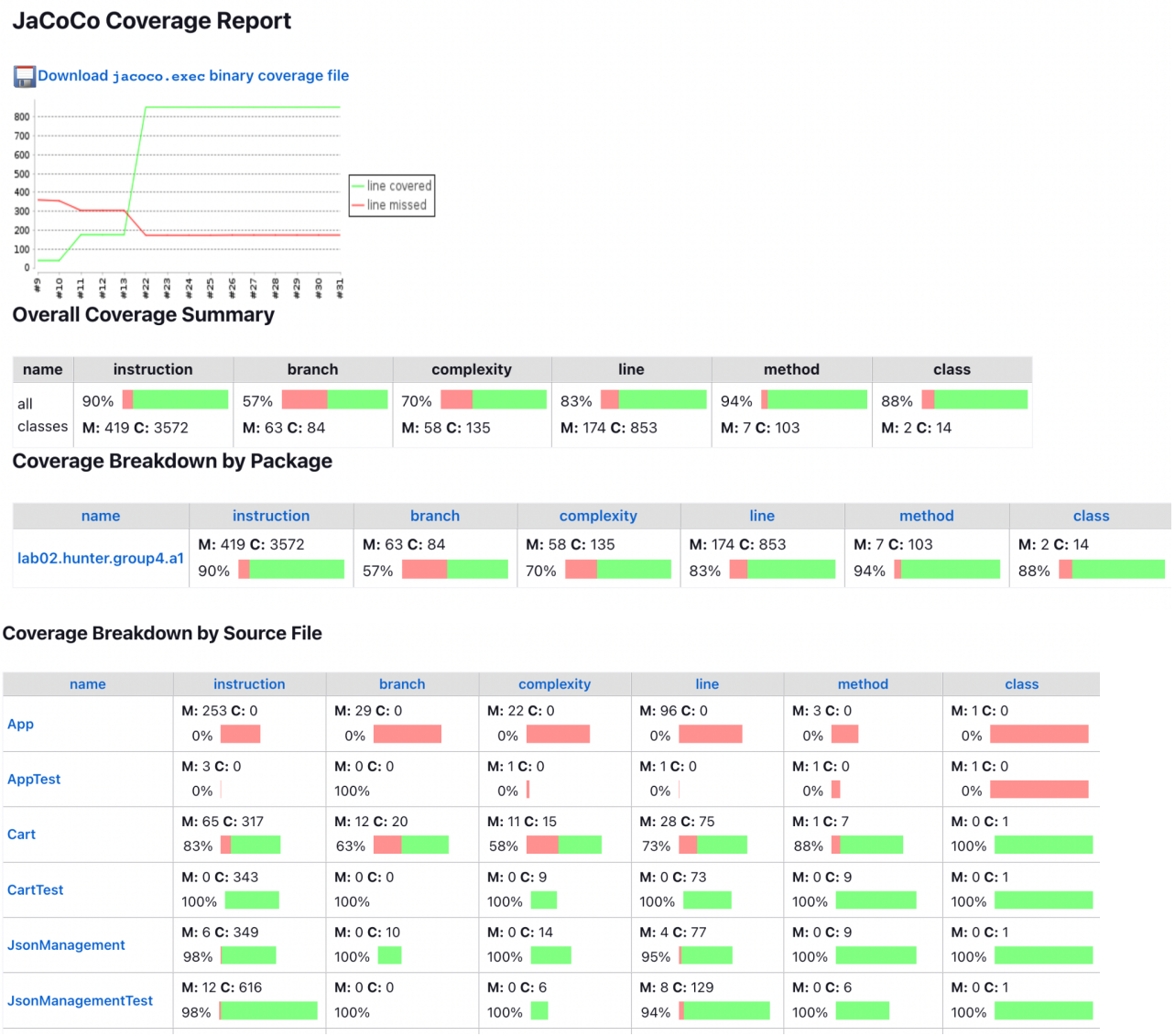


Fig 53. Jenkins Build Logs

In addition, Jenkins is able to show the coverage report for each successful builds with the JaCoCo plugin installed as seen in figure 20. Here, the report uncovers that the tests in AppTest does not cover any code. While CartTest and JsonManagementTest does cover most of the code in the respective classes. In addition to the individual report, the JaCoCo plugin allows Jenkins to also keep track of the coverage overtime as evident in figure 21.



JaCoCo Coverage Report

Download `jacoco.exec` binary coverage file

Overall Coverage Summary

name	instruction	branch	complexity	line	method	class
all classes	90% **M:** 419 **C:** 3572	57% **M:** 63 **C:** 84	70% **M:** 58 **C:** 135	83% **M:** 174 **C:** 853	94% **M:** 7 **C:** 103	88% **M:** 2 **C:** 14

Coverage Breakdown by Package

name	instruction	branch	complexity	line	method	class
lab02.hunter.group4.a1	**M:** 419 **C:** 3572 90%	**M:** 63 **C:** 84 57%	**M:** 58 **C:** 135 70%	**M:** 174 **C:** 853 83%	**M:** 7 **C:** 103 94%	**M:** 2 **C:** 14 88%

Coverage Breakdown by Source File

name	instruction	branch	complexity	line	method	class
App	**M:** 253 **C:** 0 0%	**M:** 29 **C:** 0 0%	**M:** 22 **C:** 0 0%	**M:** 96 **C:** 0 0%	**M:** 3 **C:** 0 0%	**M:** 1 **C:** 0 0%
AppTest	**M:** 3 **C:** 0 0%	**M:** 0 **C:** 0 100%	**M:** 1 **C:** 0 0%	**M:** 1 **C:** 0 0%	**M:** 1 **C:** 0 0%	**M:** 1 **C:** 0 0%
Cart	**M:** 65 **C:** 317 83%	**M:** 12 **C:** 20 63%	**M:** 11 **C:** 15 58%	**M:** 28 **C:** 75 73%	**M:** 1 **C:** 7 88%	**M:** 0 **C:** 1 100%
CartTest	**M:** 0 **C:** 343 100%	**M:** 0 **C:** 0 100%	**M:** 0 **C:** 9 100%	**M:** 0 **C:** 73 100%	**M:** 0 **C:** 9 100%	**M:** 0 **C:** 1 100%
JsonManagement	**M:** 6 **C:** 349 98%	**M:** 0 **C:** 10 100%	**M:** 0 **C:** 14 100%	**M:** 4 **C:** 77 95%	**M:** 0 **C:** 9 100%	**M:** 0 **C:** 1 100%
JsonManagementTest	**M:** 12 **C:** 616 98%	**M:** 0 **C:** 0 100%	**M:** 0 **C:** 6 100%	**M:** 8 **C:** 129 94%	**M:** 0 **C:** 6 100%	**M:** 0 **C:** 1 100%

Fig 54. JaCoCo Coverage Report for Individual Build (#31)

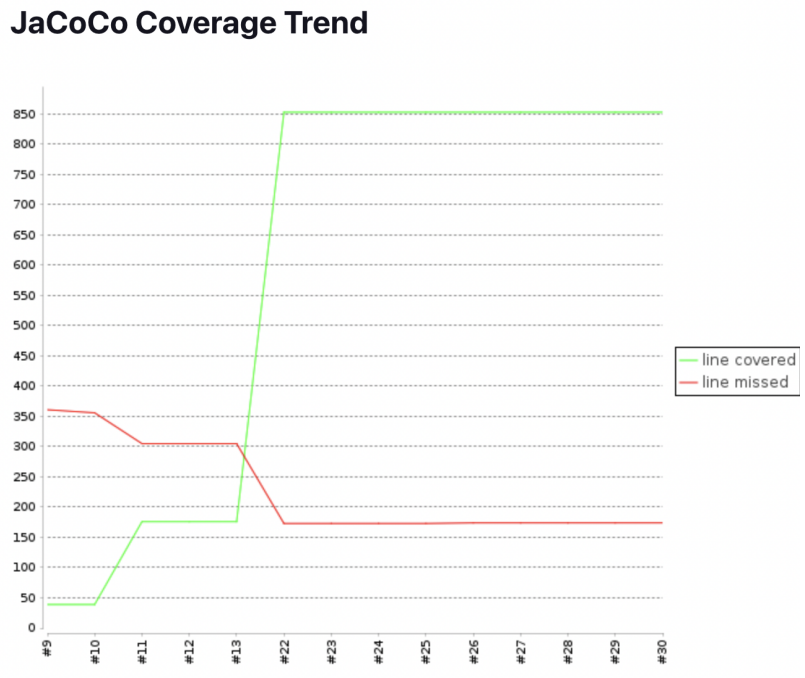


Fig 55. JaCoCo Coverage Trend

Last but not least, Jenkins provides permalinks that eases the access to the last build, last stable build, last successful build, last failed build, last unsuccessful build, and last completed build with the associated link number. This provides a good overview of the build history and the current state of the project without having to navigate through the entire build history. This is evident in Figure 22.

Fig 56. JaCoCo Coverage Trend

Common CI practices in CI undertaken by the team to ensure instant feedback and, thus code quality include frequent commits with clear commit message, and build on every commit as evident in the GitHub section of the report.

6. JUnit

6.1 Testing Frameworks

Our team has decided on JUnit 5 testing framework for our unit test as specified in the build.gradle file in figure 23, where the specific version of JUnit in use is 5.8.2. The rationale behind choosing JUnit 5 over 4 is that JUnit 5 provides additional features and flexibility for writing and executing tests. In additional, JUnit 5 has indicates a smoother integration and fewer compatibility issues for the project.

```
dependencies {
    // Use JUnit Jupiter for testing.
    testImplementation 'org.junit.jupiter:junit-jupiter:5.8.2'
    testImplementation 'org.mockito:mockito-core:3.12.4'

    // This dependency is used by the application.
    implementation 'com.google.guava:guava:31.0.1-jre'
    implementation 'com.fasterxml.jackson.core:jackson-databind:2.13.0'
    implementation 'com.googlecode.json-simple:json-simple:1.1.1'
}
```

Fig 57. build.gradle dependencies

6.2 Testing Results

Overall we have achieved a total of 78% code coverage as evident in the JaCoCo report as seen in figure 24. According to section 6.6 of the report, these tests cover a variety of scenarios, including valid, invalid, and edge cases. The assertions in the tests ensure that the methods of these classes produce the expected results. If all assertions pass, it signifies that the methods under scrutiny are functioning as expected for the given test and cases similar to the test scenarios. Section 6.7 will provide a detailed explanation of the final results achieved.

lab02.hunter.group4.a1

Element	Missed Instructions	Cov.
App		0%
Cart		82%
Menu		93%
User		87%
OrderHistory		86%
JsonManagement		98%
Order		100%
Total	389 of 1,835	78%

Fig 58. JaCoCo report on code coverage

6.3 Running Unit Testing

Unit tests can be executed through various methods, each is depended on different development environments and preferences. Modern IDEs, like IntelliJ IDEA or Visual Studio Code, have streamlined this process with intuitive integrated features. As illustrated in the figure below, users can effortlessly initiate all tests for a specific file by clicking the double arrow icon. Alternatively, individual unit tests can be run using the single arrow icon. Successful tests are conveniently marked with a green tick, providing immediate visual feedback.

Fig 59. Running test files in VS code

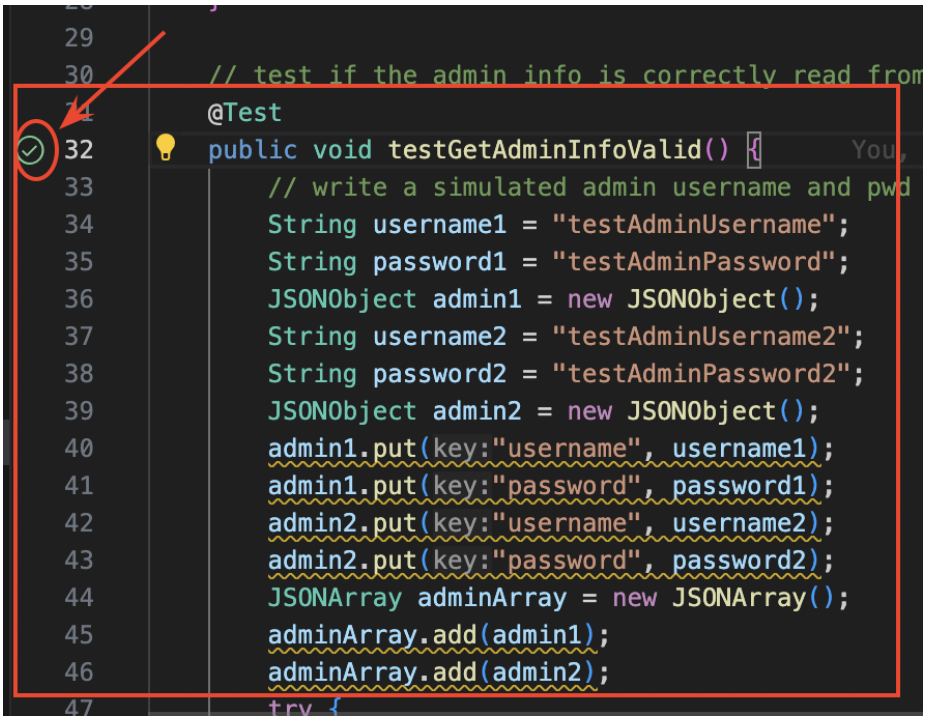
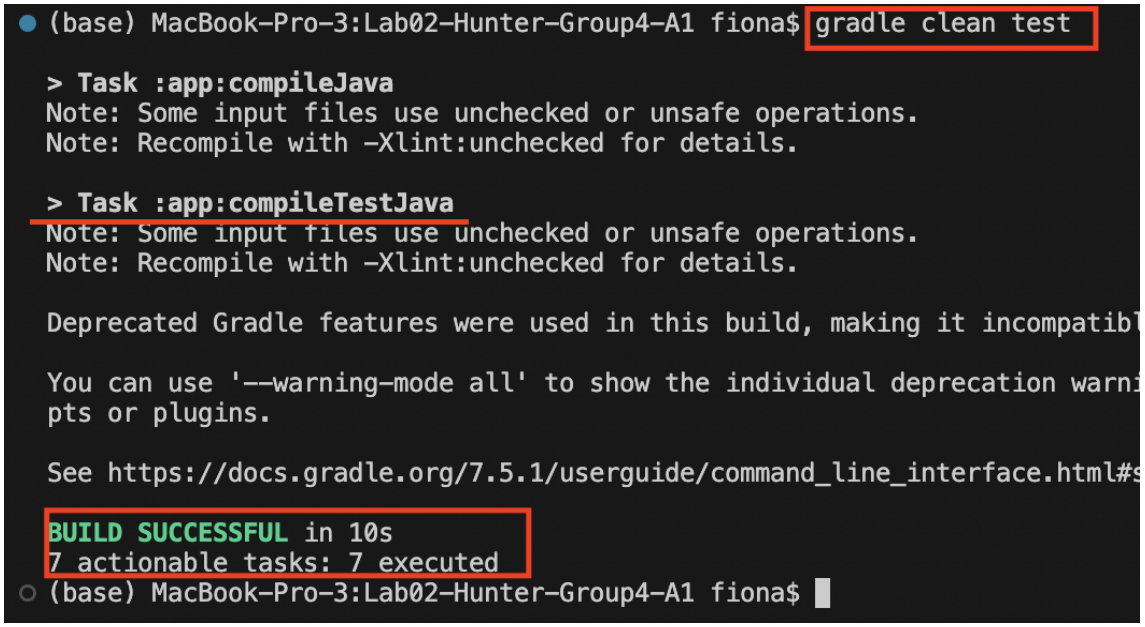


Fig 60. Running and passes one specific unit test

Another main way to execute unit tests is by leveraging the capabilities of our build tool, Gradle. Gradle is a powerful and flexible build tool used for Java and many other languages. It simplifies the process of building, testing, and deploying applications. Therefore, by executing `Gradle clean test`, all test cases present in the test folder will be executed. Alternatively, to run a single unit test using gradle, simply use the command `./gradlew test --tests <path>.<test file name>.<unit test name>.`
For instance, to run the unit test testGetAdminInfoValid() in JsonManagementTest.java, we will perform `./gradlew test --tests lab02.hunter.group4.a1.JsonManagementTest.testGetAdminInfoValid`



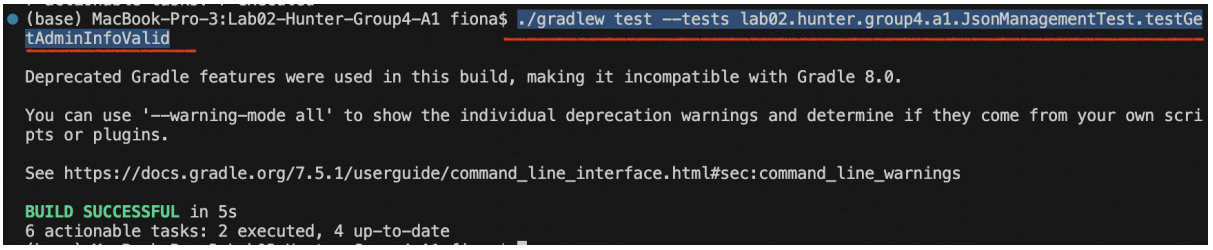


Fig 61.Running JUnit test using Gradle

6.4 Incorporating Test Coverage

Our team has incorporated JaCoCo, one of the most popular code coverage libraries for java, to measure and generate report on the code test coverage, which refers to how much of the code has been tested. This is done by integrating the JaCoCo plugin with the project through adding ‘jacoco’ within the ‘plugins’ block in our build.gradle file. After the addition, gradle clean and build needs to be rerun in order to encapsulate the changes.

Fig 62. JaCoCo plugin in build.gradle

6.5 Structure of the Test Files

The location and structure of the test files has been organised and the command `gradle init` has been executed (as seen in figure below).

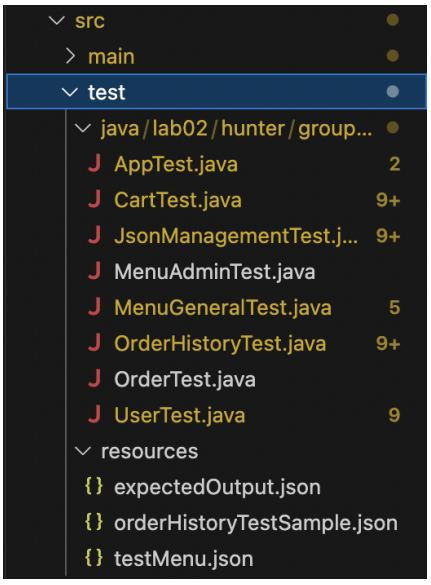


Fig 63. Location and composition of the test files

The source directory (**‘src’**) contains **‘main’** and **‘test’** directory, where ‘main’ contains all the application code and ‘test’ contains the unit test code. Within ‘test’ directory, we have **test classes** where the naming convention mirrors the main application classes (i.e. <class name>Test.java), as well as **‘resources’** directory, which includes resources exclusive for testing. This ensures that unit tests do not inadvertently modify the database or any resources crucial in operating the main classes.

6.6 Explanation of JUnit Test Cases

Class the test belong to and the name of the test	Nature of the test (correct output tests, error handling tests, boundary tests, etc.)	How was it created	what it tests	Passed/ Failed
JsonManagement				
testGetAdminInfoValid()	Correct output tests	All four testcases in json management has initiated with writing simulated information in a json file that designated for test only. This means the simulated information are expected output. The actual output is what the subject of the testing method returns. By comparing the expected (which we manually has set to be) and the actual output using assertions, we can ascertain the correctness of our methods.	Test if the method gets the correct admin information given the admin information are valid and stored in json file.	P
validGetOrderHistory()	Correct output tests		Test if the method returns the correct order history information given order details already exists in the database, which has been stored as a json file.	P
validGetMenuInfo()	Correct output tests		Test if the method return all the menu information stored in json file, including the category name and details such as item, price and descriptions.	P
testAddNewOrderValid()	Correct output tests		Tests whether the method correctly add a new order to a JSON file given the order is a valid order.	P
Order				
testGetTotalPriceValid()	Correct output tests	These tests were set up using a static setUp() method which initialises an ‘Order’ object with predefined	Test whether the attribute specific getter methods returns the associated attribute correctly	P
testGetOrderNoValid()	Correct output tests			P
testGetOrderDateValid()	Correct output tests			P

testGetOrderItemsValid()	Correct output tests	values. This object is then used in each test to verify the correctness of all the getter methods in Order Class. The expected outputs are derived from the initial set up and assertions are then used to compare whether the expected is the same as the result returned from the getter methods.		P
testGetOrderItemsInvalid()	Error handling tests			P
testGetTotalPriceInvalid()	Error handling tests			P
testGetOrderDateInvalid()	Error handling tests			P
testGetOrderNoInvalid()	Error handling tests			P
Cart				
testAddItemWithInvalidInput()	Error handling tests	1. In the setUp method, the test initialize the Cart object and set up the necessary test environment. In the tearDown method, the test clean up any resources or configurations used during the tests. 2.The test used Mockito to mock the behavior of the JsonManagement class, allowing it to control its responses and simulate different scenarios. 3.The provideInput method sets up the input stream for the tests. It converts a string into a byte stream and sets it as the input stream for the	Test if it can identify incorrect dish names and display prompts when users add Item with an incorrect dish name according to the menu.json	P
testAdjustQuantity()	Correct output tests		Test whether the shopping cart can correctly adjust the number of dishes present in the shopping cart when providing the correct dish name and quantity	P
testRemoveItemsFromCart()	Correct output tests		This test checks if the remove_items_cart method works correctly (removes an item from the cart) providing the correct dish name	P
testConfirmOrder()	Correct output tests		This test checks if the confirm_order method works correctly by	P

		Scanner. This is used to simulate user input during the tests. 4.Assertions:In each tests, assertions are used to verify that the methods being tested produce the expected results. For example, assertEquals is used to check if the quantity of an item in the cart is updated correctly. 5.The code captures the system's standard output (System.out) into ByteArrayOutputStream to check what is printed during the tests. This allows you to verify that the methods produce the expected output.	simulating user input and mocking menu items.	
testAddItemWithValidInput()	Correct output tests		Test if it can successfully realise the function of addItem in Cart class when users add Item with a correct dish name according to the menu.json	P
Menu (functionality for general user)				
testOnlyDisplayCategories()	Correct output tests	In the setUp method, the test initialized the Menu object for testing and configure it with a mocked jsonManager. Set the testMenuJsonPath = "src/test/resources/testMenu.json". Assertions.assertEquals is used to check if the displayed output	This test checks if the only_display_categories method correctly displays available categories. It simulates user input for this option and checks if the expected output is produced.	P
testDisplayCategories()	Correct output tests		This test simulates user input for selecting a category, and then it checks if the display_categories	P

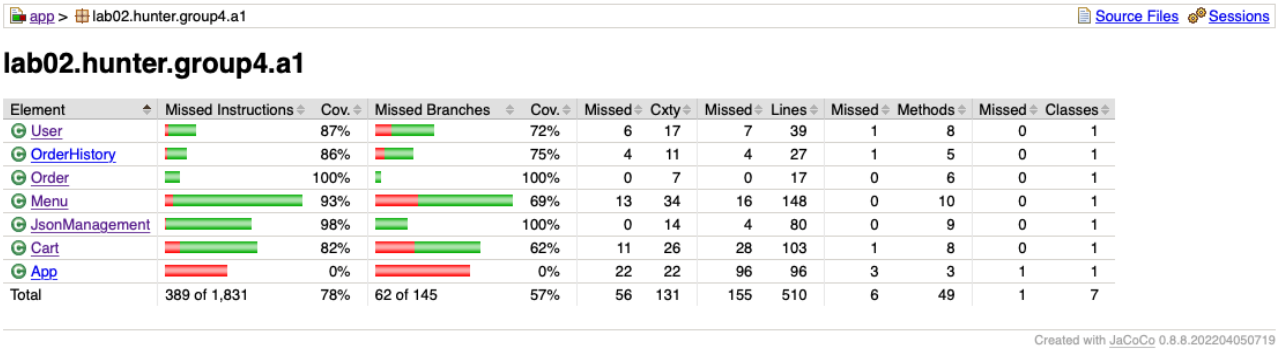
		matches the expected output. The tests set up the mock to return specific data when jsonManager.getMenuInfo is called, allowing it to control the data used in the test.	method correctly displays the items within that category based on mock data.	
testDisplayDetail()	Correct output tests		This test checks if the display_detail method correctly displays item details based on a list of mock item data.	P
testGstartOption1()	Correct output tests		This test simulates user input for option 1 (Only view Menu Categories) in the Gstart method. It verifies that the expected text is present in the output.	P
Menu (functionality for admin user)				
testAddMenuItem()	Correct output tests	The test initialises the Menu object and sets up the necessary test environment. In the tearDown method, the test cleans up any resources or configurations used during the tests. The provideInput method sets up the input stream for the tests, similar to what I did in the previous set of tests. It allows you to simulate user input during the tests. The tests use assertions to verify that the methods being tested produce the expected results. For	This test checks if the “Astart” method can successfully add a new menu item to the menu. It simulates user input for adding an item, and then it checks if the item has been added to the menu correctly.	P
testAstartInvalidChoice()	Error handling tests		This test simulates an invalid choice (choice “42”) and checks if the output contains the expected "Invalid choice" message.	P
testUpdateExistingItem()	Correct output tests		This test first adds a menu item and then tests the update functionality by simulating user input to modify the item's description and price. It checks if the item was updated correctly.	P

testRemoveOutdatedItem()	Correct output tests	example, assertEquals is used to check if items were added, updated, or removed correctly from the menu. The code captures the standard output (System.out) to check what is printed during the tests. This allows you to verify that the methods produce the expected output messages.	This test adds a menu item and then tests the removal functionality by simulating user input to remove the item. It checks if the item was removed correctly.	P
testEditCategory()	Correct output tests		This test adds a menu item and then tests the category editing functionality by simulating user input to change the category name. It checks if the category name was updated correctly in the menu.	P
Order History				
testDisplayOrderHistory()	Correct output test	The tests in this class were set up using a void setUp() method where it initialises the variable jsonFilePath and expOutputJsonPath. It then constructs JSON data to simulate an order history and writes it to expOutputJsonPath. It also initialises two objects OrderHistory and ObjectMapper. Assertions.assertEquals() was used in this class to compare the expected output with the actual output. Ensuring that the two outputs match.	This test simulates reading order from a JSON file and then calls a method to read the order history. It then checks if the output of order history matches the expected output	P
testDisplayTotalOrder()	Correct output test		This test checks if the program correctly displays the total number of orders in order history. It simulates an order and calls the display total order function. It then verifies if the output matches the expected output of the total number of orders.	P
testGetTargetedOrderExists()	Correct output test		This test ensures that the getTargetedOrder() returns the right output. It sets a	P

		Other than that, we also used Assertions.assertNotNull() to ensure that the method does not return a null when it is not supposed to. Just like in testGetTargetedOrderExists()	variable called orderNumber as an existing order number and calls the getTargetedOrder function. It checks if orderNumber exists in the order history.	
TestGetTargetedOrderDoesNotExists()	Error Handling test		This test checks if the user enters an invalid orderNumber. If the orderNumber does not exist in order history then the program is expected to display an error message of “order not found!”	P
User				
testGetUsername()	Correct output test	The tests here were set up using a static void setUp() method which initialises a User object with the predefined values. The object is then used to test the getter methods in the User class. Similar to Order History, we also used Assertions.assertEquals and Assertions.assertNotNull to ensure the methods are working properly and are returning the correct outputs. Other than that, we also use	Test whether the getUsername getter method returns the username attribute correctly	P
testGetPassword()	Correct output test		Test whether the getPassword getter method returns the password attribute correctly	P
testGetIsAdmin()	Correct output test		Test whether the getIsAdmin getter method returns the isAdmin attribute correctly	P
testRegisterAdmin()	Correct output test		Tests whether registerAdmin() can properly register a new admin and write a new admin username and password into the admin.json file.	P
testRegisterEmptyAdmin()	Error Handling test		This test simulates an invalid input for registerAdmin where the username is empty. It checks if	P

		Assertions.assertTrue to check if a condition is true. For example, in testLoginValidUser() we checked if the user is admin by calling the getIsAdmin() method on LoggedInUser. We then check if LoggedInUser is indeed an admin by using Assertion.assertTrue(). If LoggedInUser is admin then it would return true which would make it pass the testcase.	the output contains an invalid error message as it is supposed to.	
testRegisterEmptyAdminPwd()	Error Handling test		This test simulates an invalid input for registerAdmin where the password is empty. It checks if the output contains an invalid error message as it is supposed to.	P
testRegisterNonAdmin()	Error Handling test		Tests whether registerAdmin() can stop non admins from registering a new admin.	P
testLoginValidUser()	Correct Output test		This test checks whether the login() can properly identify a valid username and password and allow valid users to login.	P
testLoginInvalidUser()	Error Handling test		This test simulates an invalid input for login where the username and password does not exist in the admin.json file. It checks if the output contains an invalid error message as it is supposed to.	P-

6.7 Result Explanation



app > lab02.hunter.group4.a1 — Source Files — Sessions

lab02.hunter.group4.a1

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
User		87%		72%	6	17	7	39	1	8	0	1
OrderHistory		86%		75%	4	11	4	27	1	5	0	1
Order		100%		100%	0	7	0	17	0	6	0	1
Menu		93%		69%	13	34	16	148	0	10	0	1
JsonManagement		98%		100%	0	14	4	80	0	9	0	1
Cart		82%		62%	11	26	28	103	1	8	0	1
App		0%		0%	22	22	96	96	3	3	1	1
Total	389 of 1,831	78%	62 of 145	57%	56	131	155	510	6	49	1	7

Created with JaCoCo 0.8.8.202204050719

Fig 64. Full JaCoCo report on main classes

From the outset, the column “Element” lists the main classes in the lab-2.hunter.group4.a1 package. This includes User, OrderHistory, Order, Menu, JsonManagement, Cart and App.

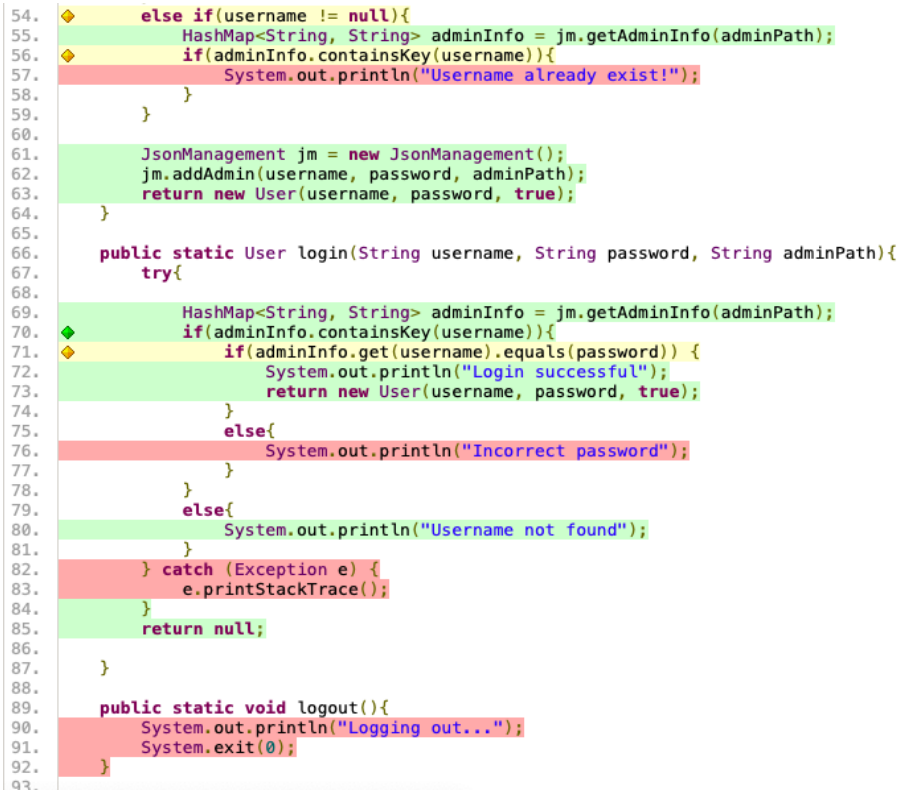
The row “Total” indicates the aggregate of all the classes. Following on, the “Missed Instructions” and “Cov” column pair provides visualised and quantitative information on the number of bytecode instructions that were not executed considering all the testcases. For example, out of 1831 instructions in total, 389 were missed, resulting in a coverage of (1831-389)/1831 = 78%.

The column pair next to the missed instruction pair is the “Missed Branches” and “Cov” column, which indicates the control flow branches that were not executed and the percentage of branches that were not covered. Other metrics such as “Missed Cxty”, “Missed Lines”, “Missed Methods”, and “Missed Classes” refers to the complexity missed, number of lines of code that were not executed, number of methods that were not executed and the number of classes that were not executed during testing.

For the purpose of substantiating the metrics, User class has a high instruction coverage of 87% and a branch coverage of 72%. Only a few methods, lines, and complexity points were missed in register_admin and login method as seen in figure below.

User

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods
User(String, String, Boolean)		100%		n/a	0	1	0	5	0	1
static {...}		100%		n/a	0	1	0	1	0	1
register_admin(String, String, Boolean, String)		94%		71%	4	8	1	16	0	1
logout()		0%		n/a	1	1	3	3	1	1
login(String, String, String)		82%		75%	1	3	3	11	0	1
getUsername()		100%		n/a	0	1	0	1	0	1
getPassword()		100%		n/a	0	1	0	1	0	1
getIs_admin()		100%		n/a	0	1	0	1	0	1
Total	16 of 129	87%	5 of 18	72%	6	17	7	39	1	8

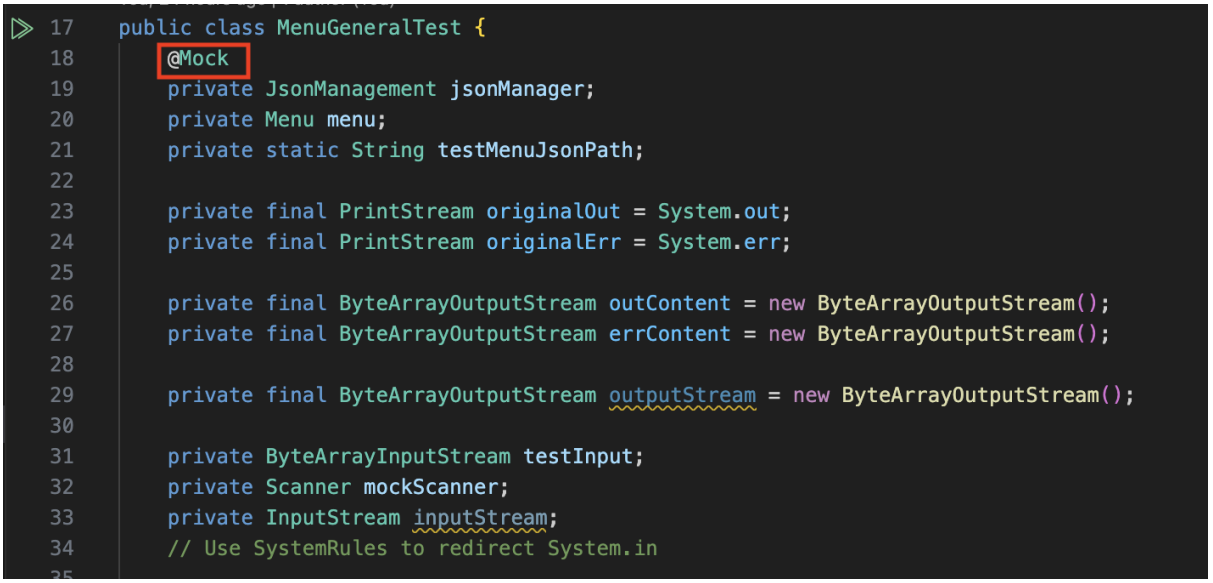


```
54.         else if(username != null){
55.             HashMap<String, String> adminInfo = jm.getAdminInfo(adminPath);
56.             if(adminInfo.containsKey(username)){
57.                 System.out.println("Username already exist!");
58.             }
59.         }
60.
61.         JsonManagement jm = new JsonManagement();
62.         jm.addAdmin(username, password, adminPath);
63.         return new User(username, password, true);
64.     }
65.
66.     public static User login(String username, String password, String adminPath){
67.         try{
68.
69.             HashMap<String, String> adminInfo = jm.getAdminInfo(adminPath);
70.             if(adminInfo.containsKey(username)){
71.                 if(adminInfo.get(username).equals(password)) {
72.                     System.out.println("Login successful");
73.                     return new User(username, password, true);
74.                 }
75.                 else{
76.                     System.out.println("Incorrect password");
77.                 }
78.             }
79.             else{
80.                 System.out.println("Username not found");
81.             }
82.         } catch (Exception e) {
83.             e.printStackTrace();
84.         }
85.         return null;
86.
87.     }
88.
89.     public static void logout(){
90.         System.out.println("Logging out...");
91.         System.exit(0);
92.     }
```

Fig 65. user class test coverage

6.8 How Results are Achieved

The test results were achieved through a combination of manual test case creation as mentioned in section 6.6 as well as automated testing tools. Mocking (as seen in figure below) was extensively used to isolate from external dependencies. Meanwhile, the use of assertions in the tests ensures that the actual output matches the expected output.



```
17  public class MenuGeneralTest {
18      @Mock
19      private JsonManagement jsonManager;
20      private Menu menu;
21      private static String testMenuJsonPath;
22
23      private final PrintStream originalOut = System.out;
24      private final PrintStream originalErr = System.err;
25
26      private final ByteArrayOutputStream outContent = new ByteArrayOutputStream();
27      private final ByteArrayOutputStream errContent = new ByteArrayOutputStream();
28
29      private final ByteArrayOutputStream outputStream = new ByteArrayOutputStream();
30
31      private ByteArrayInputStream testInput;
32      private Scanner mockScanner;
33      private InputStream inputStream;
34      // Use SystemRules to redirect System.in
```

Fig 66. Use of mock in test cases

7. Application Development Report

Clear and sensible explanation of the work carried out by the team (what, how and why) to implement the requirements of the Menu Application. This should be demonstrated through the project report and group demo including:

7.1 Group collaboration

Our group collaborated effectively to complete the Menu Application using a set of tools and strategies designed to promote efficient and productive work. Here's a detailed description of our collaboration process:

Tools Set Up
We utilized several tools to facilitate our collaboration:

Version Control: We used Git and GitHub for version control. This allowed us to work on different parts of the application simultaneously without conflicts. Each team member had their own branch where they implemented and tested their features before merging them into the main branch.

Communication: We used WeChat for regular meetings and quick updates. This helped us resolve problems quickly, share knowledge, and keep everyone up to date on the project's progress.

Work Carried Out by the Team
The team worked diligently to implement the requirements of the Menu Application. Here's an overview of our process:

Requirement Analysis: We started by thoroughly understanding the requirements. We divided these into functional and non-functional requirements, which helped us create a clear roadmap for our development process.

Design and Planning: We then designed the application's structure, keeping in mind principles of object-oriented programming, separation of concerns, and our collaboration needs. We created class diagrams and sequence diagrams to visualize the interactions between different components.

Coding: Each team member implemented their assigned features, regularly syncing their code with the main branch to ensure consistency. We adhered to good coding practices, such as writing clean, commented, and modular code.

Testing: We carried out rigorous testing of our application, both individually and as a group. We tested different scenarios to ensure the application produced the correct output and behavior.

7.2 Menu Application Structure

(2) Overview of the Menu Application design (e.g., class diagrams, sequence diagrams, or any format);

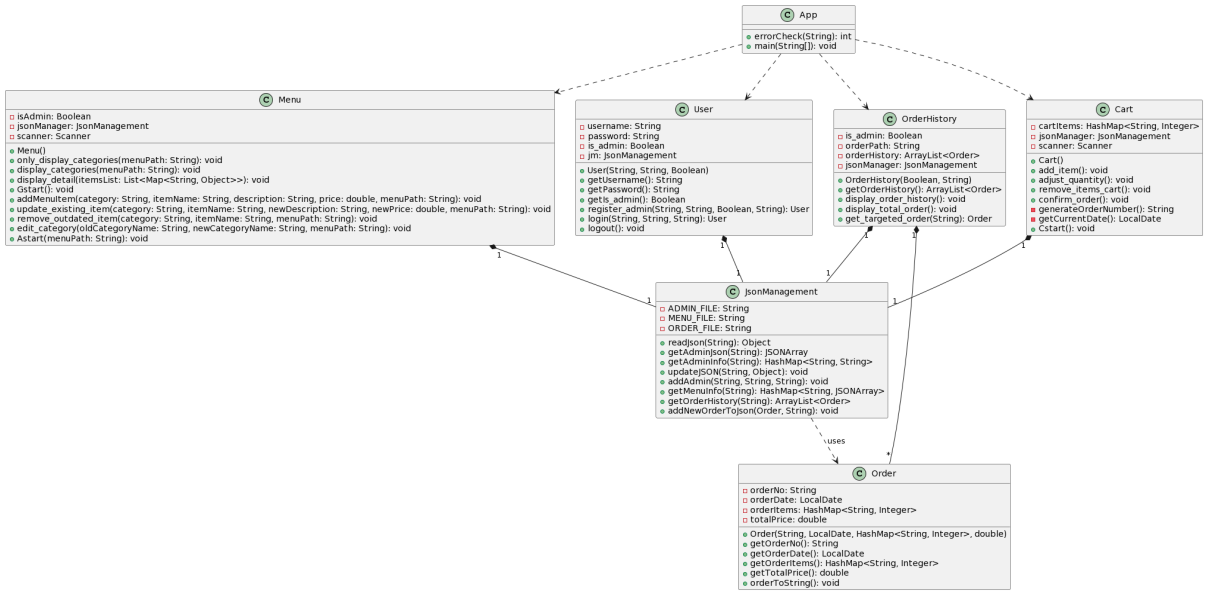


Fig 67. Menu Application Design

App: The main class that runs the application.
User: Represents a user who interacts with the application. This class includes methods for user registration, login, and logout.
JsonManagement: Handles all interactions with the JSON data, responsible for reading, updating, and managing data in the JSON files.
Menu: Represents the menu of the application. It contains methods for displaying categories and item details, adding and updating items, and starting the menu interface for both guests and admins.
Order: Represents an order made by a user. It contains order details such as order number, order date, items ordered, and total price.
OrderHistory: Manages the history of all past orders. It can display order history and specific orders.
Cart: Represents the cart of a user. It contains methods for adding and removing items, confirming orders, and starting the cart interface.

7.3 Why choose this particular structure:

We chose this particular structure for our application to separate concerns and make the code easier to maintain and develop. Each class has its own responsibilities, which are relevant to its role in the application.

User, Menu, Order, OrderHistory, and Cart classes represent different entities in the application, each with its own attributes and behaviors. This is a good example of the Object-Oriented Programming (OOP) principle, where each entity is an object.

JsonManagement class is a utility class that handles the interaction between the application and the JSON files. Keeping these methods in a separate utility class makes the code more organized and easier to maintain.

App class acts as the controller, using the other classes to execute the features of the application based on user input.

7.4 **Adherence to Functional Requirements**

The application runs in two modes: user mode and administrator mode. In user mode, the user can view menu categories and details. In administrator mode, administrators can add, update, and delete menu items, as well as edit menu categories.

When we start the application, we enter user mode. In this mode, the user can choose to view the menu or exit.

Displaying categories: When a user starts the application, they are provided with an initial menu to either view the categories or exit. If they choose to view categories, the method is called, which fetches and displays all available categories from the JSON file.display_categories(String menuPath)

Displaying items under a category: When a user selects a category, the method is called. This method fetches and displays all the items under the selected category from the JSON file.display_detail(List<Map<String, Object>> itemsList)

Adding a new menu item: When an admin chooses to add a menu item, the method is called. The admin is then prompted to enter the details of the new menu item, which is then appended to the JSON file.addMenuItem(String category, String itemName, String description, double price, String menuPath)

Updating an existing menu item: When an admin selects an item to update, the method is called. The admin is then prompted to enter the new details of the menu item, which are then updated in the JSON file.update_existing_item(String category, String itemName, String newDescription, double newPrice, String menuPath)

Removing an outdated menu item: When an admin selects an item to remove, the method is called. The selected item is then removed from the JSON file.remove_outdated_item(String category, String itemName, String menuPath)

Editing a category: When an admin chooses to edit a category, the method is called. The admin is then prompted to enter the new name of the category, which is then updated in the JSON file.edit_category(String oldCategoryName, String newCategoryName, String menuPath)

All these features strictly adhere to the functional requirements of the application. During the demonstration, every operation performed by the user or administrator was correctly reflected in the JSON file, ensuring data integrity and consistency.

Evidence:

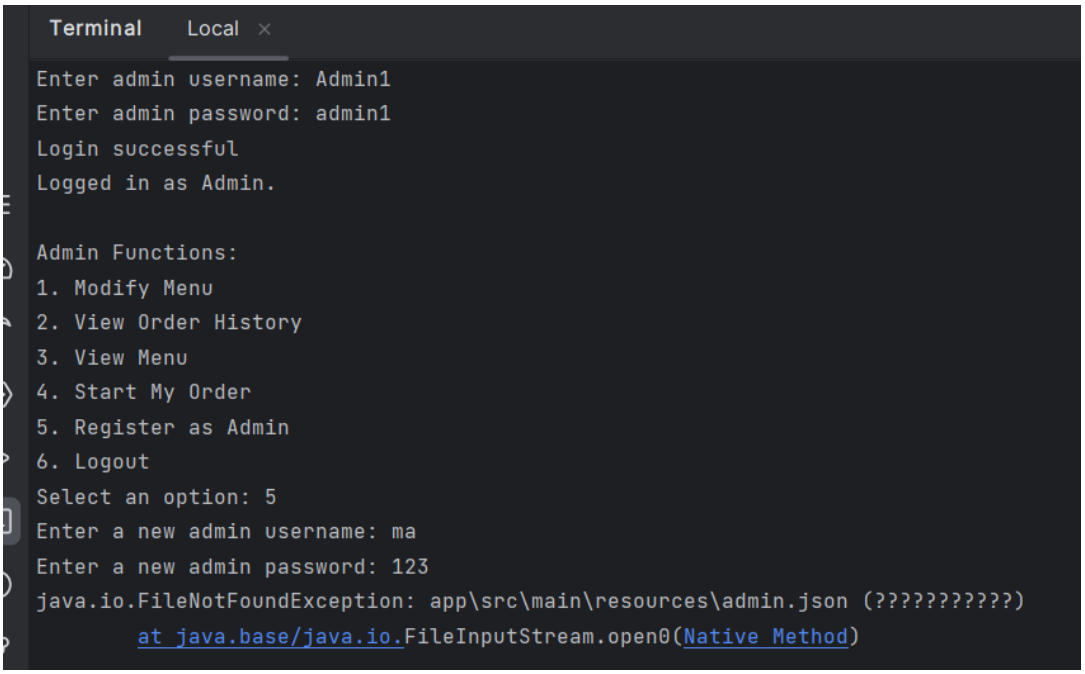


Fig 68.

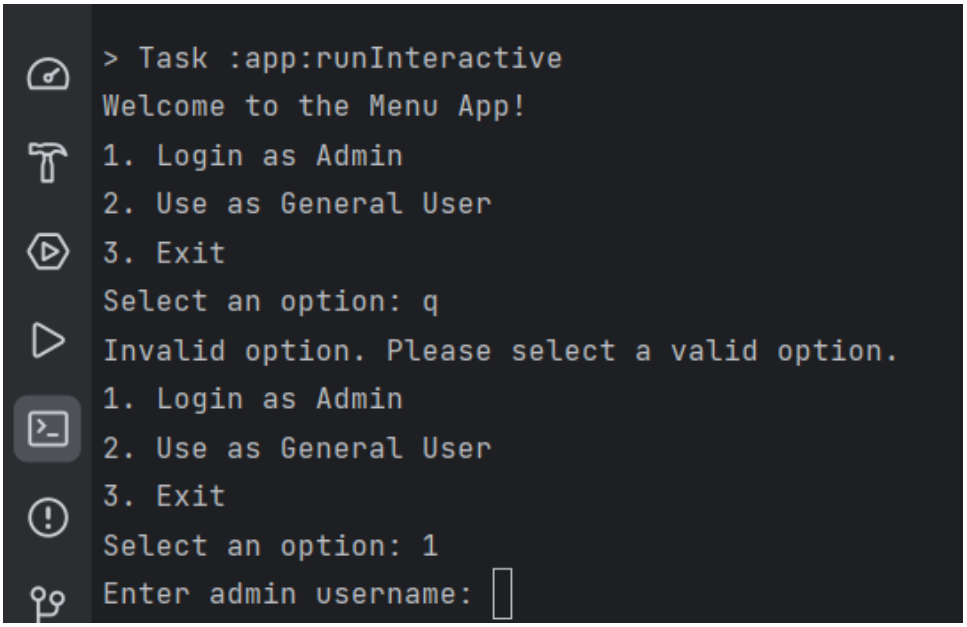


Fig 69.

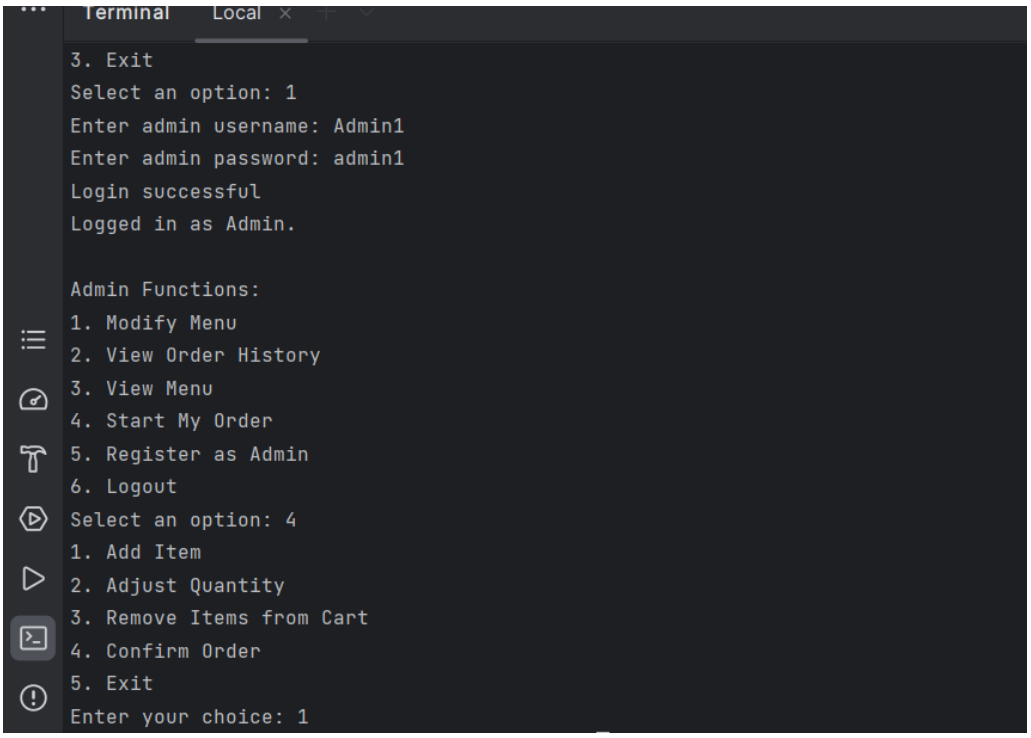

Terminal
Local
3. Exit
Select an option: 1
Enter admin username: Admin1
Enter admin password: admin1
Login successful
Logged in as Admin.
Admin Functions:
1. Modify Menu
2. View Order History
3. View Menu
4. Start My Order
5. Register as Admin
6. Logout
Select an option: 4
1. Add Item
2. Adjust Quantity
3. Remove Items from Cart
4. Confirm Order
5. Exit
Enter your choice: 1

Fig 70.

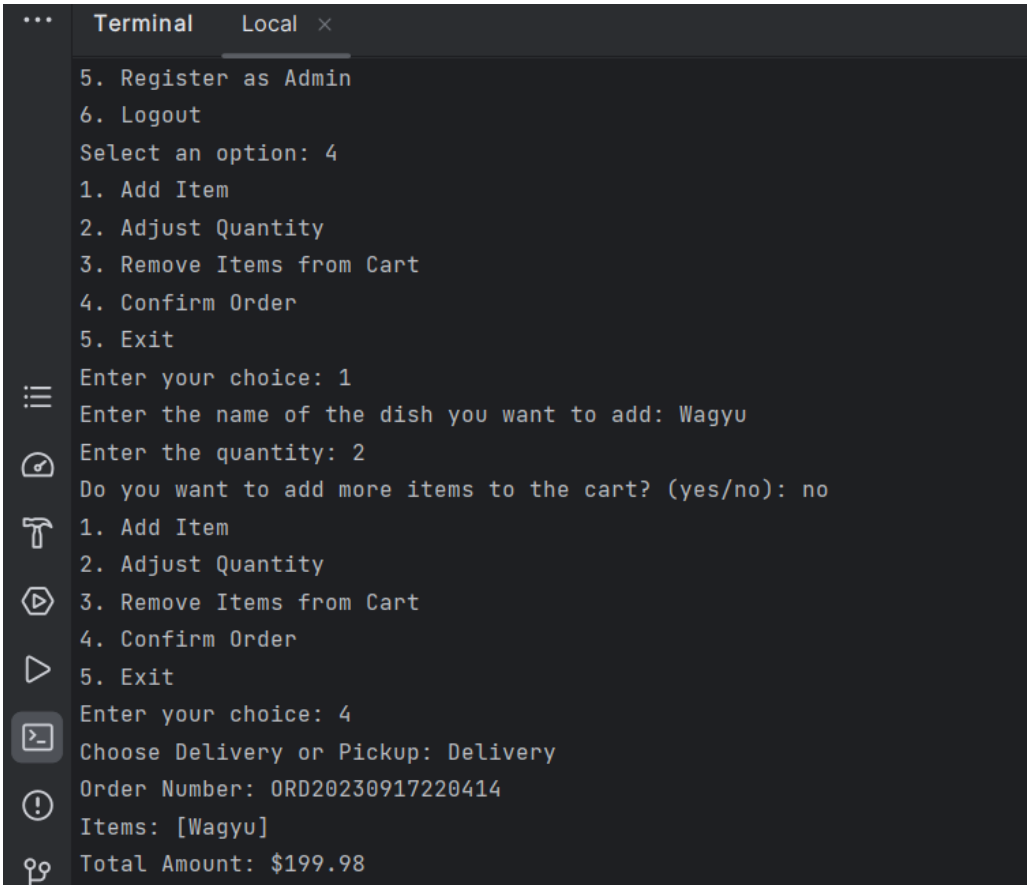

Terminal
Local
5. Register as Admin
6. Logout
Select an option: 4
1. Add Item
2. Adjust Quantity
3. Remove Items from Cart
4. Confirm Order
5. Exit
Enter your choice: 1
Enter the name of the dish you want to add: Wagyu
Enter the quantity: 2
Do you want to add more items to the cart? (yes/no): no
1. Add Item
2. Adjust Quantity
3. Remove Items from Cart
4. Confirm Order
5. Exit
Enter your choice: 4
Choose Delivery or Pickup: Delivery
Order Number: ORD20230917220414
Items: [Wagyu]
Total Amount: $199.98

Fig 71.

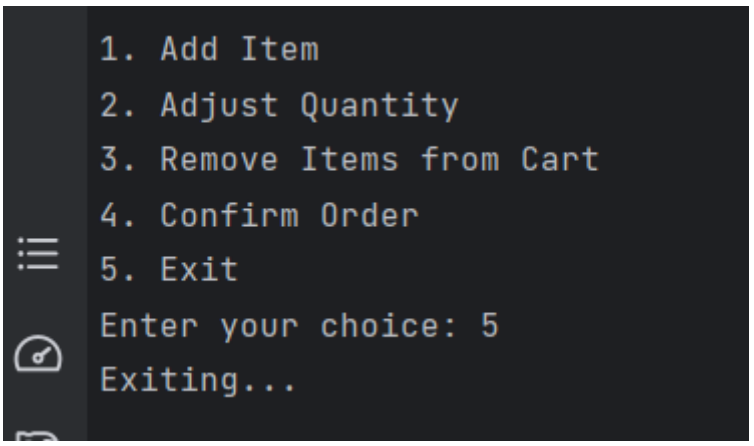
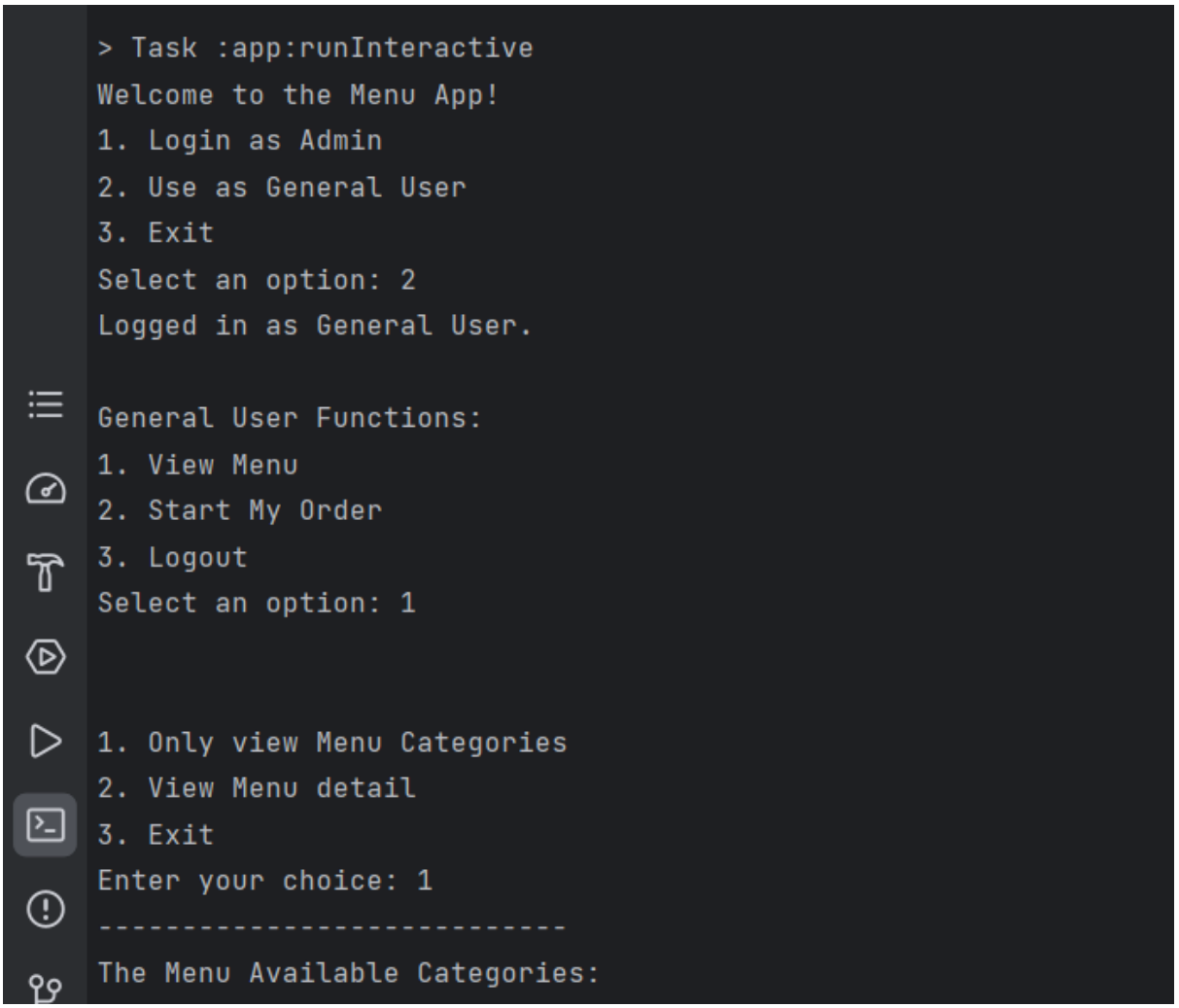

1. Add Item
2. Adjust Quantity
3. Remove Items from Cart
4. Confirm Order
5. Exit
Enter your choice: 5
Exiting...

Fig 72.


> Task :app:runInteractive
Welcome to the Menu App!
1. Login as Admin
2. Use as General User
3. Exit
Select an option: 2
Logged in as General User.
General User Functions:
1. View Menu
2. Start My Order
3. Logout
Select an option: 1
1. Only view Menu Categories
2. View Menu detail
3. Exit
Enter your choice: 1

The Menu Available Categories:

Fig 73.

7.5 Problems and Solutions

During the development process, we encountered some problems, but we adopted effective resolution strategies, here are some of them:

Question 1: When processing JSON files, data reading, writing and updating.

Solution strategy: We created a specialized JsonManagement class, which is responsible for handling all interactions with JSON files, such as reading, updating and managing data. This makes the code clearer and easier to maintain.

Problem 2: Gradle cannot find the main class and cannot run the project.

Solution strategy: We first checked the build.gradle file to ensure that we had correctly configured the main class name. If the problem persists, we check the project's file structure to make sure they match the settings in the build.gradle file.

Question 3: When compiling or running the project, sometimes Gradle will throw an error stating that some dependencies cannot be found.

Resolution strategy: We check the project's build.gradle file to confirm that all dependencies are declared correctly and that the version numbers are correct. Sometimes the problem can be due to a specific version of a library no longer being available or being renamed in the Maven Central repository or Jcenter. In addition, we will check the network connection, because Gradle needs to download dependent libraries from online sources.

7.6 Learning and improving

During the development of the project, we learned a lot about version control, team collaboration, object-oriented programming, and testing. We also understand that good requirements analysis and design planning are keys to a successful development project. We also recognize the importance of code clarity and modularity, as well as the need for rigorous testing.

For future projects, we plan to conduct requirements analysis and design planning earlier to allow sufficient time for iteration and testing. We also want to further improve our collaboration skills to work together more effectively.

Menu functionality breakdown

1. Authentication
    a. General user
        i. NOTE: Users are **not required** to log in
    b. Admin
        i. General user functionality + functionality specific to admin
        ii. Functionality: Login system
            1. Purpose: distinguish admin and general users
            2. Functionality: a simple username and password-based authentication system
2. Menu ordering
    a. Functionality: Allow general users to **select** meals from a **predefined list** of menu items available in the application
    b. Functionality: User-friendly interface that allow general users **browse** through
        i. **Different categories** of menu items
        ii. View **details** such as item names, descriptions, and prices
        iii. **Add** items to their order cart
    c. Functionality: Option to **adjust** item quantities or **remove** items from the cart if needed.
    d. Functionality: Placed order → seamless **checkout** process
        i. Selecting **delivery or pickup options**
        ii. **Confirming** the order
3. Order History
    a. Purpose: Allows users to view the order history
    b. Functionality: show order **details**

1. User class
    a. Username: String
    b. Password: String
    c. Is_admin: Boolean
    d. store_admin_info()
        i. Save user information in json file
    e. register_admin()
        i. Is admin check
        ii. Only admin can create another admins
    f. login()
        i. Enter the app
        ii. Check admin information
    g. logout()
        i. exit()

2. Menu class
    a. menu of json type
    b. Is_admin: Boolean

    General User:

    c. display_categories()
    d. display_detail(category)
        i. Item names
        ii. Description
        iii. prices

    Admin:

    e. Add_menu_item()
    f. update_existing_item()
    g. remove_outdated_item(item)
    h. edit_category(old, new)

Fig 74. Menu Functionality Breakdown I

i. Order number
ii. order date
iii. items ordered
iv. total amount

c. Stored **securely**
d. Functionality: Include options for users to **search orders based on their order number**.

4. Admin dashboard
    a. Purpose: Grants you access to manage various aspects of the application.
    b. Functionality: comprehensive overview of the **application's status**
        i. number of total orders processed
    c. Functionality: admin's special actions
        i. Perform all operations that general users can complete
        ii. **Add new menu items** (including a name, description and price for each item)
        iii. **Update existing items** (e.g. correcting prices or descriptions); and
        iv. **Remove outdated items** from the menu.
        v. Edit the categories
        vi. Create another admin

3. Cart
    a. add_item(c,i,d,p)
    b. adjust_quantity(item_name, new_quantity)
    c. remove_items_cart()
    d. confirm_order()
        i. Delivery or pickup
        ii. Generate order_number

4. Order History class
    a. Is_admin: Boolean
    b. display_order_history()
        i. Order number
        ii. order date
        iii. items ordered
        iv. total amount
    c. display_total_order()
    d. get_targeted_order (order_number)

5. JSONFileManagment Class
    a. json_to_map()
    b. Save_edited_change()

Predefined

1. Admin/admins
    a. User (user, password)
    b. User ()
2. 3 json file
    a. Admin_info.json (predefined)
    b. menu.json (predefined)
    c. Order.json

Fig 75. Menu Functionality Breakdown II