

2222124-assignment2

by Callan Birch-Williams

Submission date: 11-Dec-2022 09:32PM (UTC+0000)

Submission ID: 191900908

File name: 42481_Callan_Birch-Williams_2222124-assignment2_472250_620577085.pdf (988.47K)

Word count: 3670

Character count: 24814

UWTSD

DVLA: MSc Software Engineering (Web Development)

Combined Assessment (ACCA7000, ACCA7001, ACCA7002)

Assignment 2: Block Test

Callan Birch-Williams, 2222124

Contents

Introduction	3
Software Development Practices.....	4
Database Design.....	8
Implementation	9
Application Layer	9
Database Layer.....	10
Testing.....	12
User Story Evaluation.....	13
User Login	13
Add Booking	15
Booking Vehicle Swap	18
Contribution.....	20
Bibliography	21
Figure 1: Key Links.....	3
Figure 2: Customer User Persona Story Map (Available online via Miro)	4
Figure 3: Employee User Persona Story Map (Available online via Miro)	4
Figure 4: Example User Story	5
Figure 5: Sprint Backlog	6
Figure 6: Sprint Burndown Chart	6
Figure 7: In-progress Sprint Board	7
Figure 8: Group Database Design Concept ERD.....	8
Figure 9: Main Application REPL	9
Figure 10: Application Command Interface	9
Figure 11: Application Database ERD.....	11
Figure 12: Database Test Abstract Class	12
Figure 13: Customer Login Jira Story	13
Figure 14: Role Enum and User Class.....	13
Figure 15: Customer Login Database Query	14
Figure 16: Login Unit Test Example.....	14
Figure 17: Customer Add Booking Jira Story.....	15
Figure 18: Booking Command with Add Sub Command.....	15
Figure 19: PL/SQL Function to show Location IDs with Addresses	16
Figure 20: Booking Date Validation	16
Figure 21: Add Booking PL/SQL.....	17
Figure 22: Add Booking Test Example.....	18
Figure 23: Booking Vehicle Change Jira Story	18
Figure 24: Booking Vehicle Change PL/SQL	19
Figure 25: GitHub Contribution Report.....	20

Introduction

The purpose of this document is to evaluate and reflect on the conducting of a group assessment undertaken by myself and other students, in addition to providing a clear overview of my individual contribution to the success of the project.

Video: https://www.youtube.com/watch?v=wmZlOx5Qs_s

Repository: <https://github.com/CB2222124/rental>

Jira: <https://uwtsd-rental-group3.atlassian.net/jira/software/projects/RENTAL/boards/2>

Miro: <https://miro.com/app/board/uXjVP8q41fE=/>

Figure 1: Key Links

Software Development Practices

During the undertaking of this project, we closely adhered to the Scrum software development framework, mirroring the practices we use internally at the DVLA.

Following a brief meeting after our team was announced, it was decided that I would be assigned the role of Scrum Master. I created a new team and project on Jira to facilitate project planning and tracking.

I attended and led a user story mapping session with the team to deconstruct the brief into user stories from two user personas: customers, and employees. We decided that the consideration of a further administrative user was outside of the scope of this project. These user stories were then analysed as a group and assigned to various release stages to establish a minimum viable product.

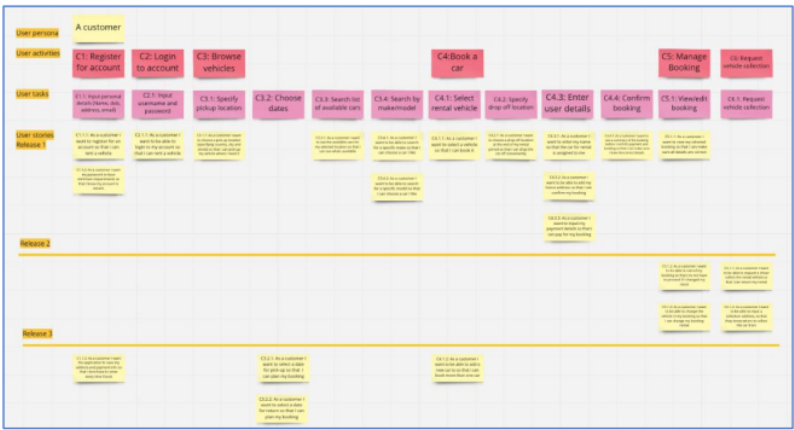


Figure 2: Customer User Persona Story Map (Available online via Miro)

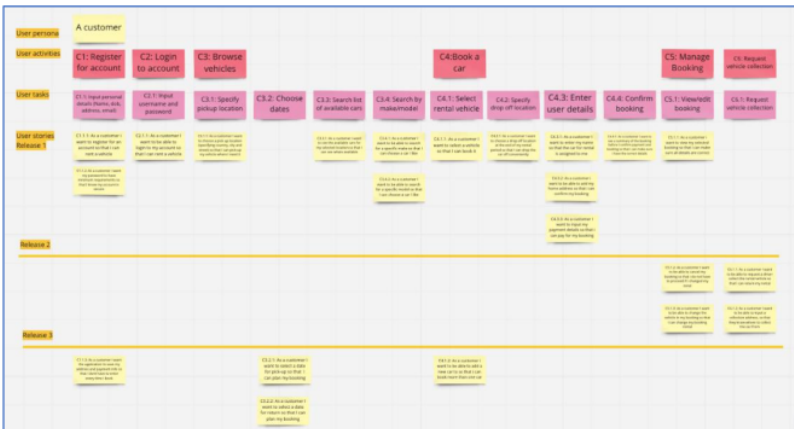


Figure 3: Employee User Persona Story Map (Available online via Miro)

A sprint planning session was held to populate the sprint backlog. As this was a self-organised project, we were also responsible for populating our own product backlog. As a team we added stories to the product backlog, in addition to their acceptance criteria (This is conventional within the DVLA).

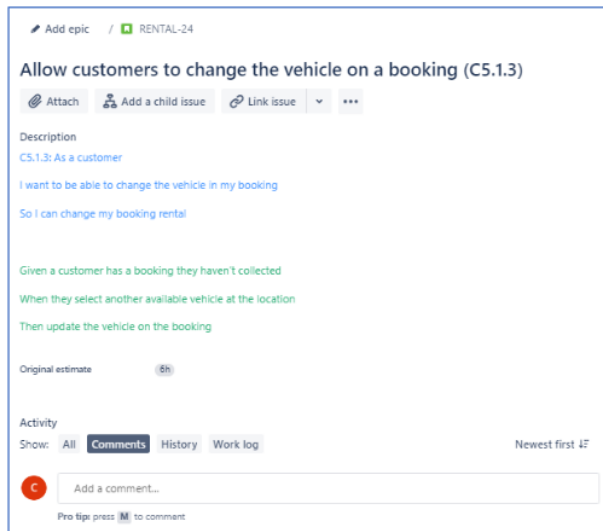


Figure 4: Example User Story

Following the population of the product backlog, user stories were timeboxed and added to the sprint. Given the difference in knowledge in relevant technical areas between team members, user stories were assigned an estimation of time by following the process of open discussion about each individual story. Consideration was given to the fact that the amount of work required for a story varied on the team member fulfilling it. Overall, the time estimates proved to be reasonable and provided the team with a good guideline for progress.

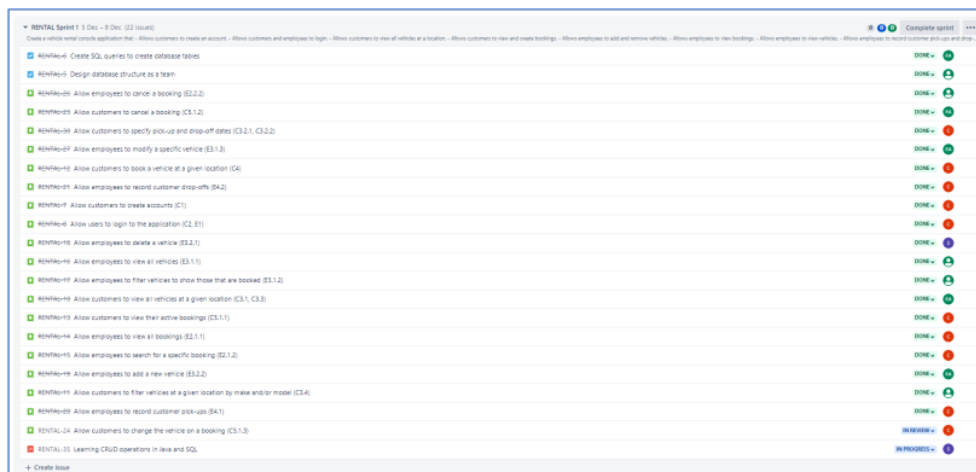


Figure 5: Sprint Backlog

Over the duration of the week-long sprint, we conducted daily stand-up meetings, which provided us with the opportunity to review outstanding stories, address issues, discuss changes and allocate any time for collaborative working (E.g., Group planning sessions and Pair Programming). We also referenced the sprints burndown chart as an indication for the likelihood of achieving our sprint goal. The reasonable progression of our burndown chart consolidated the accuracy of our initial estimations.



Figure 6: Sprint Burndown Chart

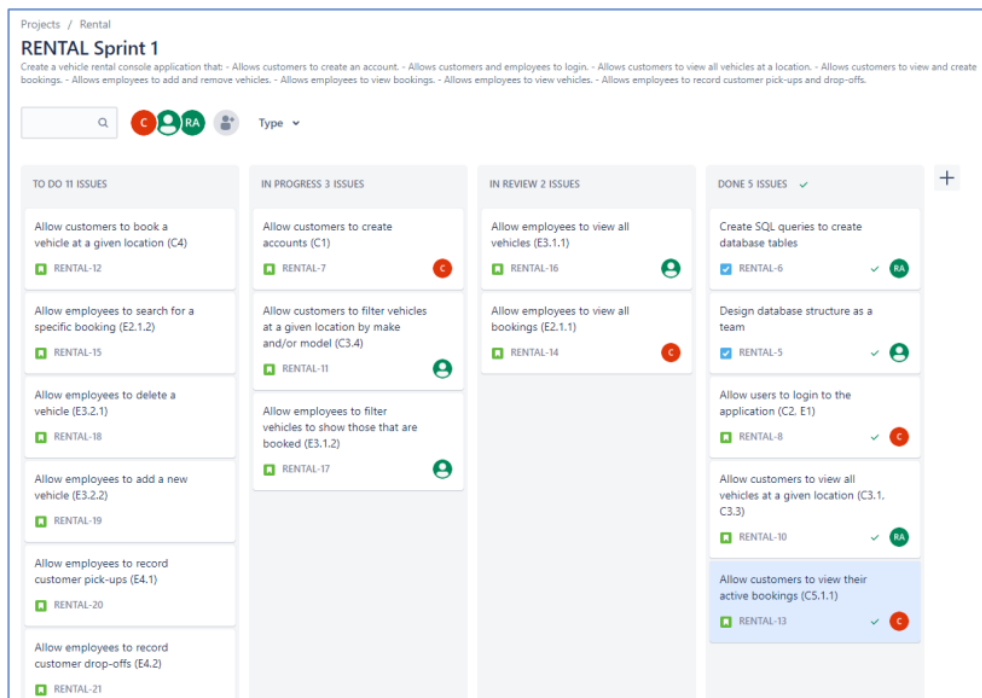


Figure 7: In-progress Sprint Board

For version control and collaborative development, I created a GitHub repository for our team. A branch protection rule was imposed on the main branch so all code changes should be applied via pull requests, subject to the review of at least one other team member. This decision helped keep team members on the same page with project progress.

In reflection to the management of this project from a software methodology perspective, I believe that by undertaking the role of Scrum Master I was able to fulfil a partial 'mentor' role to some of my peers that may have had less knowledge in certain areas of the project not yet covered by our education. By implementing agile techniques such as pair programming I was able to enable the success of my peers while still allowing them to have their own significant contribution.

Database Design

During our sprint planning session, we added two tasks to the sprint backlog that were required to be addressed prior to any implementation. These two tasks were database design as a group and creating the initial iteration of tables.

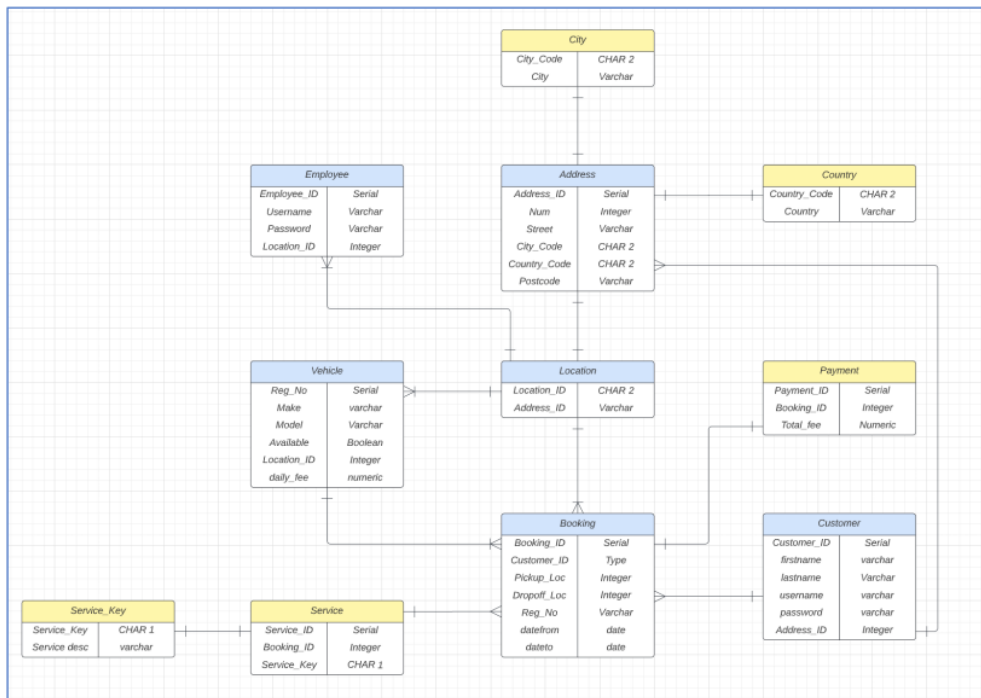


Figure 8: Group Database Design Concept ERD

The entire database conforms to 1NF and 2NF as there are only single valued attributes in each table and no partial dependencies. However, there are transitive dependencies that could be removed. For example, the Country_Code of an entry in Address is transitive dependent on the City_Code as while Country_Code and City_Code can be derived from Address_ID, Country_Code can also be derived from City_Code as a city is within a country. To remove this dependency, it would be appropriate to introduce A City_Country table which is referenced by foreign key in Address.

Implementation

Application Layer

The Java application layer to this project was built on top of a Read–Eval–Print–Loop (REPL) that processed user input to execute a specific command. This structure is preferred to a standard flow application with branching logic and interdependencies that make development as a team difficult.

By choosing a command driven approach, team members were able to write self-contained components for each command within the application. This not only made collaborative working easier but also facilitated feature-driven development (FDD), by allowing team members to focus on implementing a specific feature without having to handle integration with other parts of the codebase. This played to the strengths of the team given I was the only member familiar with Object-oriented programming principles in Java.

```
private static void startREPL() {
    Scanner scanner = new Scanner(System.in);
    CommandParser parser = new CommandParser();
    while (true) {
        System.out.print("> ");
        String input = scanner.nextLine();
        Command command = commands.get(parser.getCommand(input));
        if (command != null && command.isAvailable()) command.execute(parser.getArguments(input));
        else System.out.println("'" + input + "' is not a valid command. Try 'commands'.");
        System.out.println();
    }
}
```

Figure 9: Main Application REPL

Shown above is the code responsible for driving the application. User input is evaluated, and a specific command is executed. A command in the context of the application is a class that implements the interface shown below. Team members then created their own classes that implemented the command interface, allowing them to be executed during user input evaluation.

```
public interface Command {
    void execute(LinkedHashMap<String, String> args);
    boolean isAvailable();
    String getDescription();
}
```

Figure 10: Application Command Interface

Given an opportunity to revisit this project, there are a few improvements I would make:

1. Firstly, I would define some easier distinction in code for whether a command is a sub command, as while the structure of the application is relatively easy to understand now, at scale it might be a concern with the existence of customer commands, employee commands and sub commands all under the same interface.
2. I would make arguments a more significant part of the user's interaction with the system. For example, a user logging in might just type "login -user <username> -password <password>", rather than being prompted for inputs after executing the command. This would better conform to standard CLI design. Additionally, a "-help" argument could be implemented by every command to aid user application interaction.
3. While partially a byproduct of having the business logic of the application in both Java and PL/SQL, database raised exceptions are not communicated well to the application user, but rather output a generic message with the database exception in parentheses. It would be suitable in future iterations to implement a layer of abstractions that elegantly handles database exceptions to produce clear output to the user.

In terms of further work, this project has been conducted in a component-based way and could easily be built upon in various directions:

1. Convert the application to a true CLI tool by making use of a dedicated command line application library like picocli [1], coupled with GraalVM [2], which can provide Java with ahead-of-time compilation for fast start-up (Necessary for a production CLI application).
2. Harness this application as middleware between the database layer and some web-based frontend application.
3. Build a desktop application with a graphical user interface.

Database Layer

The database application layer to this project uses PostgreSQL within a Docker container. SQL scripts stored in the application project root are moved to created containers and ran to instantiate the database with tables, functions, and dummy data. This made it easy to add and track changes to database design and application by avoiding the common pitfall of local state and dependencies. It had the additional benefit of providing a way to use a common state for testing, something that will be covered in further detail in the Testing section of this document.

The database table schema was initially constructed to adhere to the relationships outlined in the ERD created during the group design session. Over the lifecycle of this project, iterative changes were made to satisfy the MVP.

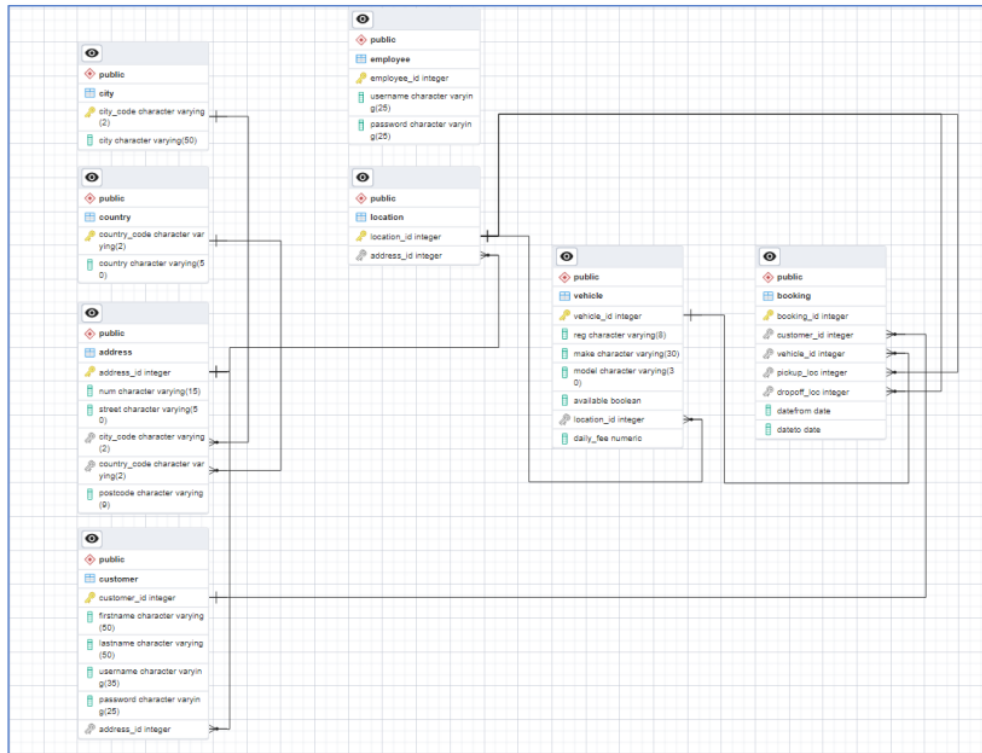


Figure 11: Application Database ERD

Due to the nature of the project being conducted over just one week by a team with varying experience, database operations were written in both Java and PL/SQL to play to the strengths of individual team members.

Given time to revisit this project, some of the various changes to the database application layer that we would make include:

- Making use of triggers to create audit tables for operations on vehicles and bookings. This is something that was attempted, but ultimately fell out of scope of the project.
- Implementing all database operations in PL/SQL, separating business logic from the application layer.
- Consider the implications of race conditions in database operations written in PL/SQL.
- Keep PL/SQL functions in their own related files and copy a directory of functions into the container, rather than one long file of functions (I did attempt to achieve this but was not able to do this correctly with Docker within the time allocated for this project).
- Revisit database design.

Testing

To facilitate database testing, I created an abstract parent class to wrap database unit tests that would reset the state of the database between tests and provide child tests with a connection to the database. This allowed team members to write unit tests for database operations without the overhead of establishing a common database state and connecting to it.

```
@TestInstance(TestInstance.Lifecycle.PER_CLASS)
public abstract class CommandTest {

    private static final String TEST_URL = "jdbc:postgresql://localhost:5432/rental";
    protected Connection connection;

    @BeforeAll
    public void connectDatabase() throws SQLException {
        Properties properties = new Properties();
        properties.setProperty("user", "postgres");
        properties.setProperty("password", "postgres");
        connection = DriverManager.getConnection(TEST_URL, properties);
    }

    @BeforeEach
    public void resetDatabase() throws SQLException {
        CallableStatement create = connection.prepareCall("{call createTables()}");
        CallableStatement populate = connection.prepareCall("{call populateTables()}");
        create.execute();
        populate.execute();
    }

    @AfterAll
    public void disconnectDatabase() throws SQLException {
        connection.close();
    }
}
```

Figure 12: Database Test Abstract Class

To remove the dependency on a running container from testing, I experimented with building and creating a container as part of the test process. However, I was not able to get this working within the time allocated for this project. If I were to revisit this project in the future, I would explore the benefits of dedicated database testing tools such as Testcontainers for Java [3] or pgTAP [4].

User Story Evaluation

This section will cover the lifecycle of a select few user stories I fulfilled to provide some insight on the FDD approach taken with individual command components.

User Login

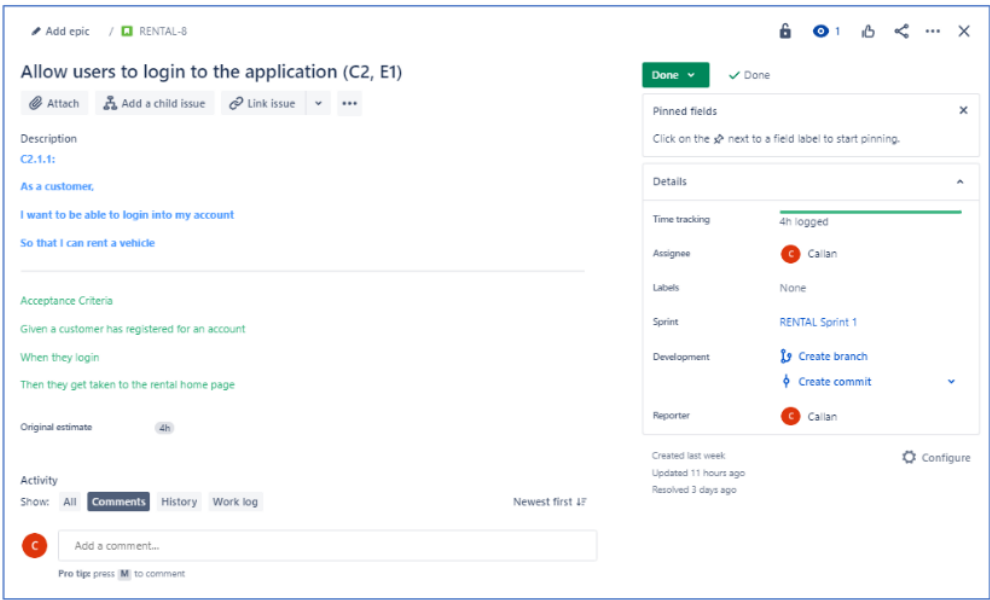


Figure 13: Customer Login Jira Story

To facilitate customer login, there first must be some concept of 'logged in' within the application. I constructed a user class and role enumeration responsible for holding the current user's role and ID, which is instantiated on start-up and globally visible within the application. This is used throughout the application to evaluate state, as other commands will check for user role when evaluating if they should be available. For example, the user login command is only available if the current user's role is 'NONE'. In reflection, the application user could have been passed to each command on execution rather than putting it in the global scope.

```
public enum Role {  
    NONE, CUSTOMER, EMPLOYEE  
}
```

```
public class User {  
    private Role role;  
    private int id;  
    ...  
}
```

Figure 14: Role Enum and User Class

The login command is registered at start-up with a keyword and an instance of the command. When the command is executed by the user using the keyword, they are prompted for input which is stored in local variables. If the argument key 'employee' is present, then the command will look for the user's login in the employee table, otherwise it will search the customer table. This was done to avoid the situation of not knowing which table to evaluate where the username is present in both the employee table and customer table. This may be an indication of opportunity to refactor the database design rather than implementing application-level workarounds.

```
public int login(String table, String username, String password, Connection connection)
    throws SQLException, NoSuchElementException {
    PreparedStatement statement = connection.prepareStatement(
        "SELECT " + table + "_id FROM " + table + " WHERE username = ? AND password = ?");
    statement.setString(1, username);
    statement.setString(2, password);
    ResultSet result = statement.executeQuery();
    if (!result.next()) throw new NoSuchElementException();
    return result.getInt(1);
}
```

Figure 15: Customer Login Database Query

Unit testing was conducted for the function presented previously. As the test class extends the abstract command test class discussed previously, there is no need to write code to establish a clean database state or connect to the database. Assertions are made to ensure that the login ID retrieved is the correct expected ID in relation to the clean database state. Additionally, an assertion is made that the function will throw a specific exception if no login is found with the provided information.

```
public class LoginTest extends CommandTest {

    @Test
    public void validLogin() throws SQLException {
        LoginCommand loginCommand = new LoginCommand();
        int id = loginCommand.login("customer", "user101", "letmein", connection);
        assertEquals(1, id);
    }

    @Test
    public void throwOnInvalidLogin() {
        assertThrows(NoSuchElementException.class, () -> {
            LoginCommand loginCommand = new LoginCommand();
            loginCommand.login("customer", "user101", "letmein2", connection);
        });
    }

    ...
}
```

Figure 16: Login Unit Test Example

Add Booking

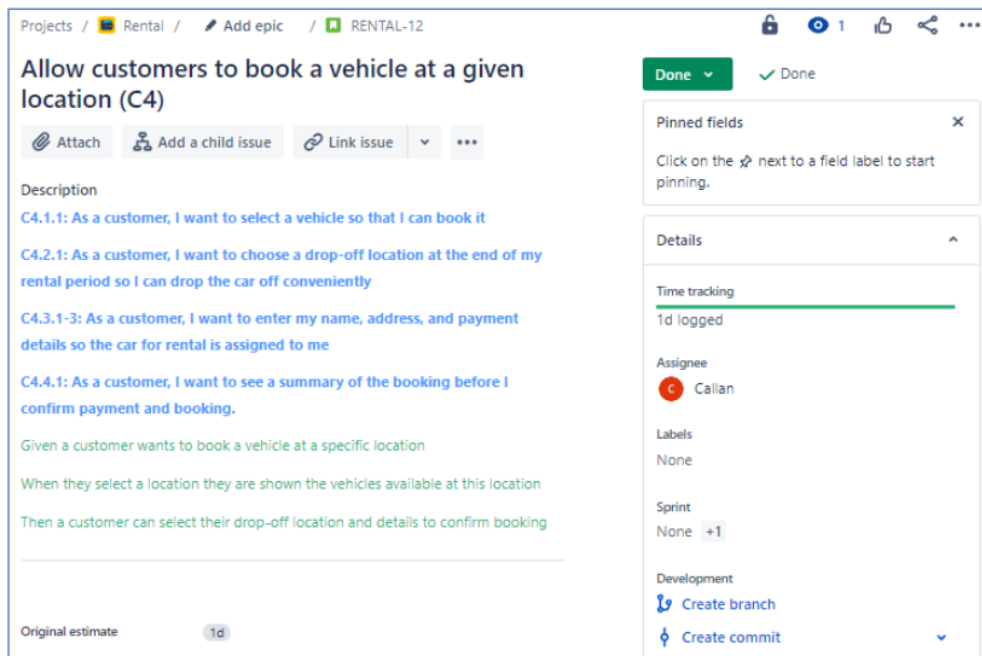


Figure 17: Customer Add Booking Jira Story

The 'add booking' command is executed by the customer through a parent 'booking' command, by entering 'booking -add' when prompted for user input in the application loop.

```
public class BookingCommand implements Command {
    @Override
    public void execute(LinkedHashMap<String, String> args) {
        ...
        switch (command) {
            case "new" -> {
                if (Application.user.getRole() == Role.CUSTOMER) {
                    new BookingAddCommand().execute(args);
                }
            }
        }
        ...
    }
}
```

Figure 18: Booking Command with Add Sub Command

When the command is executed, the customer is prompted to provide the vehicle ID that they would like to book (This can be found by using the 'vehicles' command). A command used to view location IDs with their corresponding addresses is automatically executed. The customer is then expected to input a location ID as the drop-off location.

```
CREATE FUNCTION getLocationsWithAddresses()
RETURNS TABLE (
    location_id integer,
    num varchar,
    street varchar,
    city_code varchar,
    country_code varchar,
    postcode varchar
)
AS $$ BEGIN Return Query
SELECT location.location_id, address.num, address.street, address.city_code, address.country_code,
address.postcode
FROM location
INNER JOIN address ON location.address_id = address.address_id;
END;
$$ language plpgsql;
```

Figure 19: PL/SQL Function to show Location IDs with Addresses

The customer is then prompted to specify pick-up and drop-off dates, which are validated based on the following rules:

- Both dates are valid and represented as strings in yyyy-MM-dd format.
- The pick-up date is not before the current day.
- The drop-off date is not before the pick-up date.

```
public boolean validateBookingDates(String pickup, String dropoff) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat("yyyy-MM-dd");
    sdf.setLenient(false);
    Date pickupDate = sdf.parse(pickup);
    Date dropoffDate = sdf.parse(dropoff);
    Date today = sdf.parse(LocalDate.now().toString());
    return pickupDate.compareTo(today) >= 0 && dropoffDate.compareTo(pickupDate) >= 0;
}
```

Figure 20: Booking Date Validation

In reflection, while the validation of the pick-up and drop-off dates relative to each other and the current day make sense to be within the add booking command, validation of the data itself should instead be extracted to a helper validation class. This is an improvement that should be made application-wide for validation of user input.

Finally, the customer is asked to confirm their intention by providing confirmation input. In future iterations, the customer would be expected to enter further details like payment information just prior to this operation (And for the supporting business logic to be implemented). Following confirmation, a new booking is added.

```
CREATE FUNCTION addBooking(input_customer_id integer, input_vehicle_id integer, input_dropoff integer,
input_date_from date, input_date_to date)
RETURNS VOID
AS $$ BEGIN
IF NOT isVehicleAvailable(input_vehicle_id) THEN
    RAISE 'Vehicle is not available for booking';
END IF;
INSERT INTO booking (
    customer_id,
    vehicle_id,
    pickup_loc,
    dropoff_loc,
    datefrom,
    dateto,
    with_customer
)
VALUES (
    input_customer_id,
    input_vehicle_id,
    (SELECT location_id FROM vehicle WHERE vehicle_id = input_vehicle_id),
    input_dropoff,
    input_date_from,
    input_date_to,
    false
);
END;
$$ language plpgsql;
```

Figure 21: Add Booking PL/SQL

Unit tests were created for both input date validation and the booking add function. I was responsible for creating unit tests for the booking add function. The tests I chose were for standard addition success, throws on invalid IDs and importantly throws on an unavailable vehicle being specified. Note that while the customers are only shown available vehicles, they can still enter the IDs of unavailable ones. Therefore, we must account for this. In reflection, this may be a possible area for improvement on the application layer, by checking user input against the vehicles shown before wasting a call to the database layer.

```

public class AddBookingTest extends CommandTest {

    @Test
    public void addNewBooking() throws SQLException {
        BookingAddCommand bookingAddCommand = new BookingAddCommand();
        bookingAddCommand.addBooking(1, 1, 1,
            Date.valueOf("2023-01-01"), Date.valueOf("2023-01-08"), connection);
        PreparedStatement statement = connection.prepareStatement(
            "SELECT * FROM booking WHERE customer_id = ? AND vehicle_id = ?");
        statement.setInt(1, 1);
        statement.setInt(2, 1);
        ResultSet resultSet = statement.executeQuery();
        assertTrue(resultSet.isBeforeFirst());
    }

    @Test
    public void throwsOnUnavailableVehicle() {
        assertThrows(PSQLException.class, () -> {
            BookingAddCommand bookingAddCommand = new BookingAddCommand();
            bookingAddCommand.addBooking(1, 3, 1,
                Date.valueOf("2023-01-01"), Date.valueOf("2023-01-08"), connection);
        });
    }
}
...

```

Figure 22: Add Booking Test Example

Booking Vehicle Swap

The screenshot shows a Jira story page for 'Allow customers to change the vehicle on a booking (C5.1.3)'. The story is in the 'To Do' state and is assigned to Callan. The description includes a user story and acceptance criteria. The right sidebar shows details like time tracking (6h logged), assignee (Callan), labels (None), sprint (RENTAL Sprint 1), and development options (Create branch, Create commit). The bottom section shows activity with a comment input field.

Story Title: Allow customers to change the vehicle on a booking (C5.1.3)

Description: C5.1.3: As a customer
I want to be able to change the vehicle in my booking
So I can change my booking rental

Acceptance Criteria:
Given a customer has a booking they haven't collected
When they select another available vehicle at the location
Then update the vehicle on the booking

Original estimate: 6h

Activity: Show: All Comments History Work log. Newest first 17

Comments: Add a comment...

Details:
Time tracking: 6h logged
Assignee: Callan
Labels: None
Sprint: RENTAL Sprint 1
Development: Create branch, Create commit
Reporter: Callan

Metadata:
Created last week
Updated yesterday
Resolved yesterday
Configure

Figure 23: Booking Vehicle Change Jira Story

Similar to the 'add booking' command discussed previously. The 'change booking' command is executed by the customer through a parent 'booking' command, by entering 'booking -change' when prompted for user input in the application loop.

The user is then prompted to enter a booking ID and the ID of the vehicle they would like to change their booking to before executing the operation.

In order for the operation to be successful, the following criteria must be met:

- The specified booking must belong to the customer.
- The current vehicle must not have already been picked up by the customer.
- The new vehicle is available and at the same location.

```
CREATE FUNCTION updateBookingVehicle(input_booking_id integer, input_vehicle_id integer, input_customer_id
integer)
RETURNS INTEGER
AS $$ BEGIN
UPDATE
  booking b
SET
  vehicle_id = input_vehicle_id
WHERE
  input_booking_id = booking_id
  AND input_customer_id = customer_id
  AND with_customer = 'false'
  AND isVehicleAvailable(input_vehicle_id)
  AND EXISTS (SELECT * FROM vehicle WHERE vehicle_id = input_vehicle_id AND location_id = b.pickup_loc);
RETURN COUNT (*) FROM booking WHERE booking_id = input_booking_id AND vehicle_id = input_vehicle_id;
END;
$$ language plpgsql;
```

Figure 24: Booking Vehicle Change PL/SQL

Contribution

The purpose of this section is not to discredit the relative efforts of others, but rather provide an objective overview of the work contributed by myself towards the completion of this project.

When considering the time estimates for each user story, I was responsible for 47.5/92, or 51.6% of the hours allocated to the sprint, as opposed to the expected 25% when considering the project was conducted by 4 people. In addition to this, a large proportion of other contributions were reviewed and altered by myself during integration due to my familiarity with Java principles not yet covered within our education (This is clearly documented in the codebase where I am listed as a secondary author to the main contributor for a file).

Testing was not recorded directly on the sprint board, but rather remaining time outside of the sprint was allocated for it. As such consideration for my contribution towards testing can be verified by the discussion in the Testing section of this document, in addition to checking the author tag for each test component in the codebase.

Other contributions to the completion of this project include:

- Scrum Master responsibilities.
- Creating the Jira team and project.
- Creating the GitHub repository.
- Creating a template application for team members to build upon.



Figure 25: GitHub Contribution Report

Bibliography

- [1] "picocli - a mighty tiny command line interface," [Online]. Available: <https://picocli.info/>. [Accessed 11 12 2022].
- [2] "GraalVM," [Online]. Available: <https://www.graalvm.org/>. [Accessed 11 12 2022].
- [3] "Testcontainers for Java," [Online]. Available: <https://www.testcontainers.org/>. [Accessed 11 12 2022].
- [4] "pgTAP: Unit Testing for PostgreSQL," [Online]. Available: <https://pgtap.org/>. [Accessed 11 12 2022].

FINAL GRADE

92/100

GENERAL COMMENTS

Instructor

Sw Des Eval (30):27

Database Eval (30):27

Code & Test (30):29

Reflection (10):9

Feedback:
=====

Good design and discussion but could have expanded a little on the story estimation approach and sprint planning disucssions.

More evaluation of the design was expected, this is very brief.

Excellent code effort using a number of techniques that have not been introduced in the course.

It would have been nice to have seen more specific reflection on challlenges that you faced.

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8

PAGE 9

PAGE 10

PAGE 11

PAGE 12

PAGE 13

PAGE 14

PAGE 15

PAGE 16

PAGE 17

PAGE 18

PAGE 19

PAGE 20

PAGE 21