

ELE 1300155 111222

by Elanwy Evans

Submission date: 11-Dec-2022 04:20PM (UTC+0000)

Submission ID: 191867577

File name: 42119_Elanwy_Evans_ELE_1300155_111222_472250_1161833530.docx (2.83M)

Word count: 5625

Character count: 28696



Prifysgol Cymru
Y Drindod Dewi Sant
University of Wales
Trinity Saint David

University of Wales Trinity Saint David
School of Computing

11/12/2022

ACCA7000/ ACCA7001/ ACCA7002

Assignment 2: Block Test

Software Dev and Testing/Agile SEP/Database Systems

By Elanwy Evans (Student ID 1300155)

Table of Contents

<i>Introduction.....</i>	<i>3</i>
Links	3
<i>Discussion.....</i>	<i>4</i>
Initial Project Planning.....	4
Refinement / Sprint Planning.....	7
My Individual Contribution	10
<i>Reflection</i>	<i>20</i>
<i>Conclusion</i>	<i>23</i>

Introduction

This report will discuss, evaluate and reflect on the design and development of a console-based car rental company application. It will include a detailed discussion of my individual work and an evaluation of my relative contribution to the sprint.

Links

- Miro Board link:
https://miro.com/app/board/uXjVP8q41Qk=?share_link_id=731786714918
- Jira board link: <https://t1-car-rental-co.atlassian.net/jira/software/projects/TCR/boards/1/backlog>
- Git Repo link: <https://github.com/elanwyevans/car-rental-company#java-setup-instructions>

NOTE 1: To run the program, please follow the instructions in the Readme file on GitHub.

NOTE 2: My database procedures and functions were originally written and stored in the 'EE-register-or-log-into-account.sql' file within the 'register_or_log_into_account' folder. This file still exists, however for ease of use, everyone's sql scripts have been combined in a single file called 'ddl-script.sql' in the FINAL-SCRIPTS folder. As stated in the Readme file, the 'combined-insert.sql' script should be executed first, followed by the 'ddl-script.sql' script.

Discussion

Initial Project Planning

As a team, we set up a Microsoft Teams meeting to discuss our plan of action for the sprint. This didn't follow a traditional Agile convention (i.e. no 3-amigos session) as we didn't have a product owner to consult with and didn't have the time to conduct all the ceremonies. Firstly, we agreed on a 12:00am cut off point on 10/12/22 where we would stop writing code, so that we would have approximately 48 hours to write our reflective reports and demo our MVP.

We discussed our interpretation of the coursework brief and designed an MVP for the working console application. The application requirements were vague when referring to the end user. It specified a list of requirements that the 'customer' should be able to perform, however this list included features that would usually only be available to administrators e.g. 'book a vehicle for maintenance or repair'. As the segregation of 'customers' and 'administrators' was not explicitly requested and we were limited on time, we decided that we would create an application where all features would be universally accessible by everyone. We acknowledged that, in reality, a company would likely want different access levels for employees/administrators and customers, e.g. where only administrators would have permissions to manage vehicle status. However, it would have been a lot of work to implement a feature that was essentially based on an assumption, therefore we decided to prioritise the features that were explicitly requested in the brief.

We created a user story map for the sprint. When creating the map, we initially decided that our MVP (release 1) would include every feature listed in the brief (figure 1). We gave each activity/epic, user task and user story a unique identifier so that they would be easier to reference.



Figure 1: Initial User Story Map Created by Team 1

We created a Jira board and added the user stories for release 1 to the product backlog ready to be estimated (Figure 2).

▼ Backlog (32 issues)		0000	Cr
TCR-2	I1.1 As a user, I want to be able to see a list of all the vehicles in the inventory and their statuses so that I can manage the inventory	TO DO	▼
TCR-3	I2.1 As a user I want to be able to add a vehicle to the inventory so that I can keep the company inventory up to date when vehicles are bought	TO DO	▼
TCR-4	I2.2 As a user, I want to be able to remove a vehicle from the inventory so that I can keep the inventory up to date when vehicles are sold.	TO DO	▼
TCR-5	I3.1 As a user I want to specify a start and end date so that I can track when the vehicle is unavailable due to repairs	TO DO	▼
TCR-6	I3.2 As a user, I want to specify what type of repair / maintenance a vehicle is going for so that I can keep track of the repairs and maintenance needed	TO DO	▼
TCR-8	E1.1 As a user I want to be able to select an available vehicle of my choice so that I can add to it my booking		
TCR-9	E1.2 As a user I want to be able to add my selections and proceed with booking so that I can book myself a vehicle		
TCR-10	E1.3 As a user I want to be able to enter my name and home address So that it can be on my bookings		
TCR-11	E1.4 As a user I want to be able to enter date and time for my vehicle collection So that I can book it for duration of my requirement and see the vehicles available on that date		
TCR-12	E2.1 As a user I want to be able to enter my payment details so that I can pay for my booking		
TCR-13	E2.2 As a user I want to be able to finalise my booking So that I can get confirmation of pickup		
TCR-14	E3.1 As a user I want to be able to view the booking I have made So that I can know more details about my booking		
TCR-15	E5.1 As a user I want to be able to cancel my booking So that If I do not require vehicle anymore I can cancel it		
TCR-16	F1.1 As a user, I want to be able to add pick up / drop off locations (specifying country, city and street), so the customer can see an up to date list of available options		
TCR-17	F1.2 As a user, I want to be able to remove pick up / drop off locations (specifying country, city and street), so the customer can see an up to date list of available options		
TCR-18	H2.1 As a user, I want to be able to record when a vehicle has been picked up so that inventory and customer booking records are up to date		
TCR-19	H3.1 As a user I want to record a vehicle drop off So that I can track that a vehicle was returned by the customer		

Figure 2: Jira Product Backlog

I added an 'in-review' column to the Jira sprint board as the coursework brief specified that all code must be reviewed (figure 3).

TO DO	IN PROGRESS	IN REVIEW	DONE ✓
 You haven't started a sprint You can't do anything on your board because you haven't started a sprint yet. Go to the backlog to plan and start a sprint. Go to Backlog			

Figure 3: Jira Sprint Board Columns

We then discussed how we were going to lay out the database. Arthur had already created a high-level design of the database before starting the project, so we reviewed and discussed his design, ensured the tables were appropriately normalised, made some changes and created a low-level database design as a team (figure 4). We collectively decided that city and country would need to be in separate tables to avoid repetition of city entries. We also created audit tables to allow us to implement trigger functions when certain transactions were made. We acknowledged that we might not have time to do this, however we decided it would be easier to include them at the beginning and not use them, as opposed to adding them in at a later date.

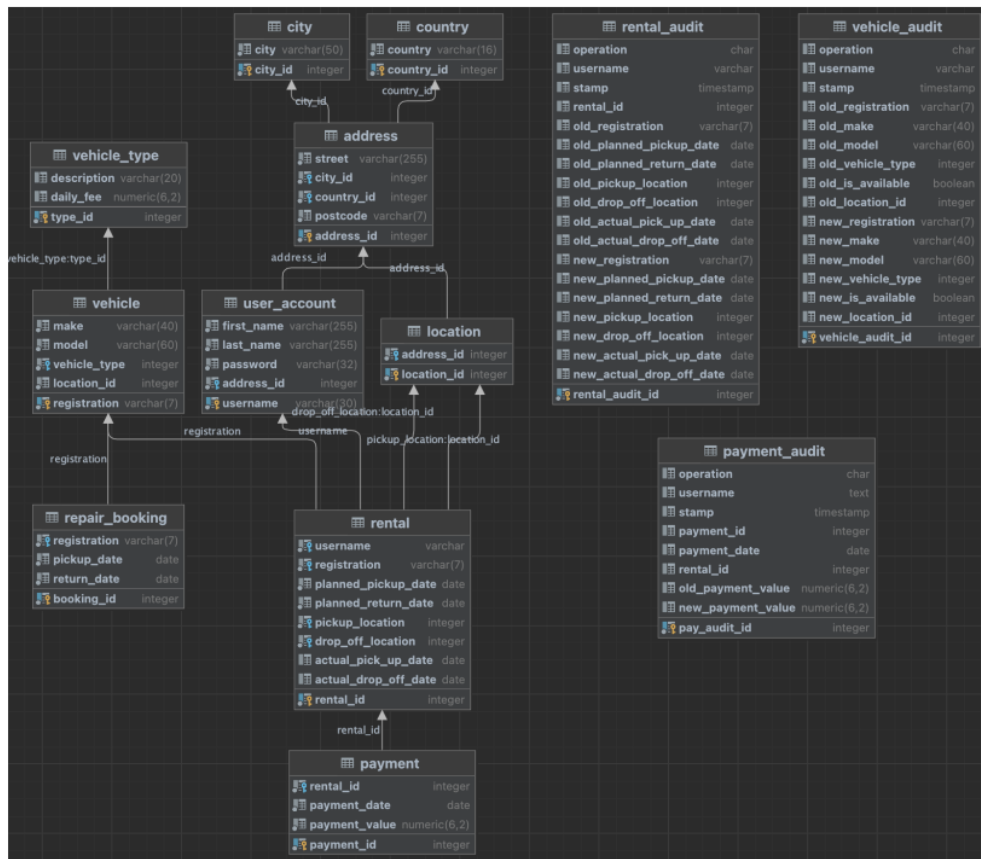


Figure 4: Entity Relationship Diagram Representing Finalised Database Design

Some of the changes I suggested included:

- removing some tables (employee table & repair type table) as the brief did not mention different user or repair types
- removing the 'is_available' field from the vehicle table
- removing the 'user_id' primary key from the user_accounts table and just using the username itself as the primary key as it is unique
- introducing constraints (such as setting username, password, first name, last name, first line of address, city, country etc to be a certain length)
- changing naming convention to snake case and ensuring consistency across all entities/fields
- Removing storage of card numbers in the payments table as this would breach PCI DSS

Ideally, it would have been better if all team members were involved in the high-level database design, however due to time constraints, we decided to use Arthur's design as a foundation to build the database.

I set up a private git repository and put rules in place so that a pull request could not be merged to the main branch until all discussions were resolved and it had 2 approvals. As it was a private repository and we didn't have organisational licences, the rules could not be enforced, so we had to trust that everyone would not merge to main until the conditions were met. This was a risk as there were no controls in place to prevent someone accidentally merging to main and breaking the code. However, the only alternative was to make the repository public, which was not an appropriate option as anyone could have accessed our work and claimed it as theirs.

We ended our initial planning session by setting up daily 15-minute stand up meetings for the duration of the sprint and a 60-minute refinement meeting (figure 5). In a real production environment, retrospective and review sessions would be added to the sprint, however we decided against this as we had a very limited amount of time to produce a functional MVP.

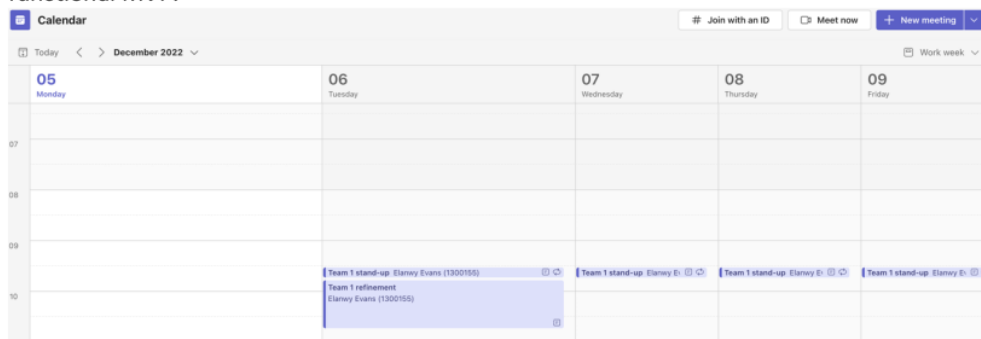


Figure 5: Teams Calendar with Meetings Scheduled

Refinement / Sprint Planning

As a group, we estimated the user stories using planning poker on Miro. This was difficult as we didn't have a sprint velocity to base our estimations on. Also, everyone in the team had different levels of programming experience so the scores were quite variable at times. We managed this by explaining the reasoning behind our scoring and adjusted our individual scores accordingly, and/or settled on the median score.

As we worked our way through the user story map, we realised that we likely weren't going to complete all the user stories we'd originally planned to do for our MVP. As a result, we reorganised the board by moving some of the stories from release 1 section to the release 2 section (figure 6).

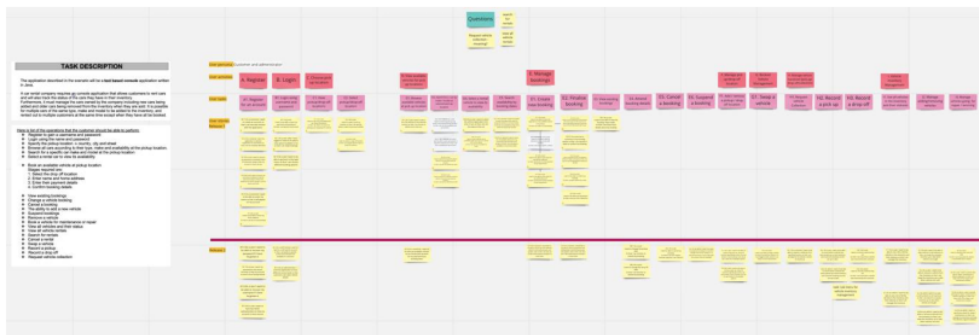


Figure 6: Final User Story Map

We decided to focus on creating a functional application for a customer to be able to browse and rent a vehicle (i.e. the left hand side of the map). We agreed that the vehicle management features (i.e. the right hand side of the map, which include actions that would typically be performed by an administrator) would need to be part of a later release. As a result, we also had to change the user stories in the product backlog to reflect these changes.

We agreed that each of us would focus on a different feature to try and avoid overlap/conflicts and duplication of work. The user stories assigned to me are shown in figure 7.

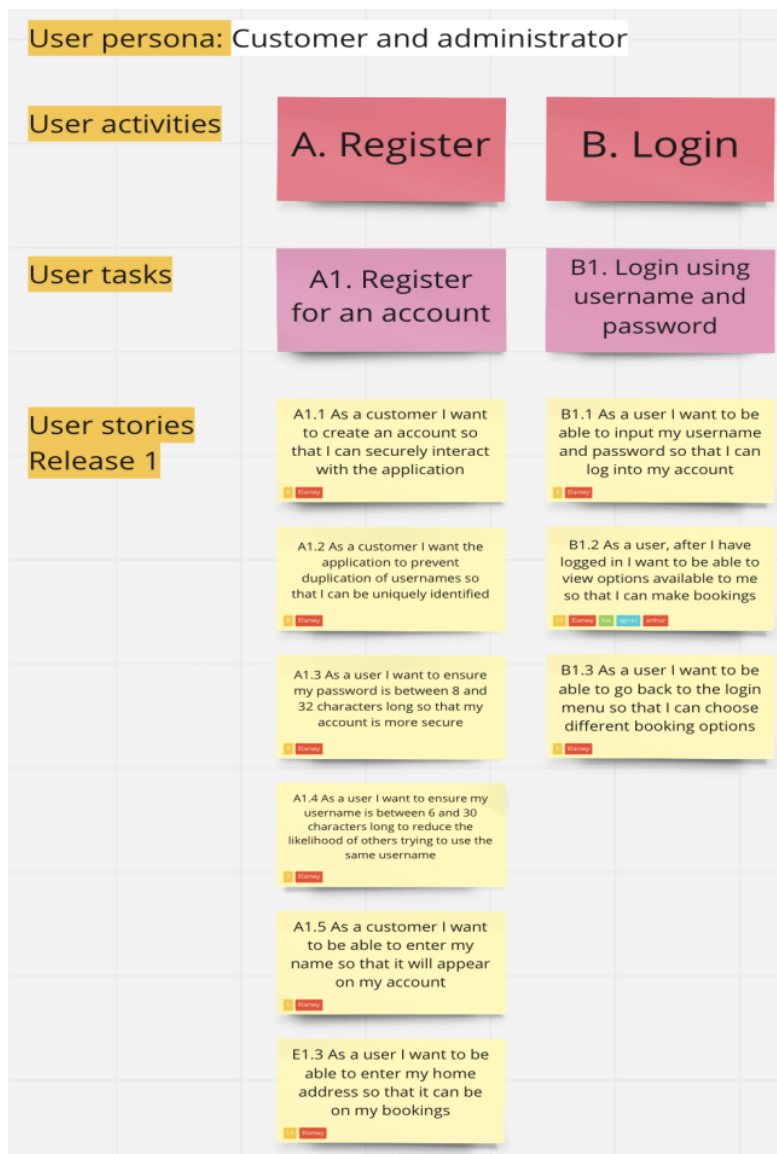


Figure 7: User Stories Assigned to Me

Because we didn't have much time, we collectively agreed that instead of creating user tasks (child issues) for the user stories as a team, we would individually create user tasks for the specific stories we were working on. This wasn't an ideal approach as it is important to gather input from a variety of perspectives to highlight potential issues that might be overlooked otherwise, but we took the risk so that we could make a start on the application ASAP.

My Individual Contribution

Initial Java Setup

I created the initial Java file on my feature branch 'feature/setup-connection-to-database' (figure 8) which included:

- a function to connect to the database ('getConnection')
- a basic 'menu' in the Main.java class for logging in/registering for an account, so that other members of the group would know where to add their functions
- generic functions (enterValueForInt & enterValueForString & checkIfResultSetEmpty) that could be used by everyone to avoid repetition of code
- a function to set up a mock database connection (setUpMockDBConnection) and a corresponding unit test (testConnectionToDatabase())

Feature/setup connection to database #3

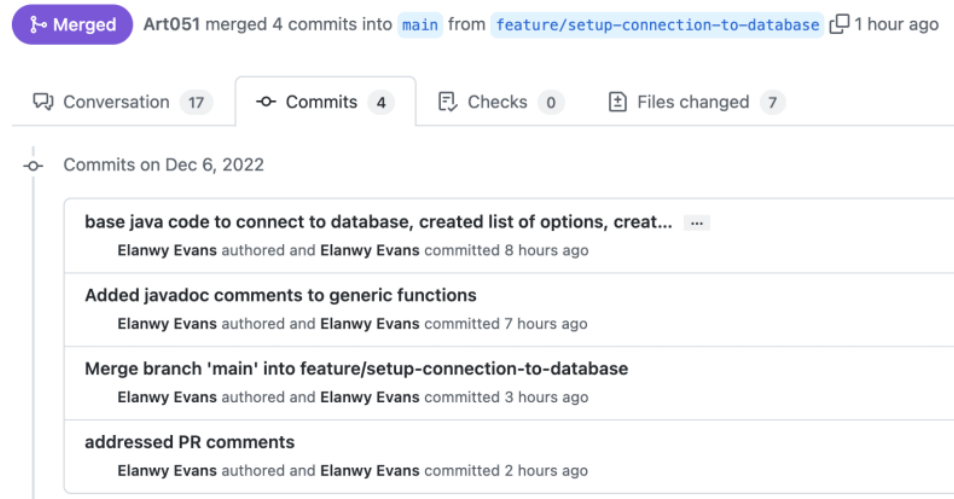


Figure 8: Commits Made to the 'feature/setup-connection-to-database' Branch

I pushed it to Git as soon as I could so that everyone could utilise the same folder structure and the 'boilerplate' functions. We initially encountered technical difficulties which slowed us down slightly, however after consulting as a team, we were able to identify and resolve the issue relatively quickly by adjusting the Java folder structure.

Create an Account Feature

The following list includes some of the user stories and associated user story tasks (child-issues) I worked on as part of the 'create account' feature:

- User story A1.1 As a customer I want to create an account so that I can securely interact with the application
 - A1.1.1: Create a table in the database to store account details
 - A1.1.2: Create function in Java that takes user input for username and password
 - A1.1.3: Create procedure in postgres to update account table with new usernames and passwords
- A1.2 As a customer I want the application to prevent duplication of usernames so that I can be uniquely identified
 - A1.2.1. Create function in sql to check if the username entered already exists
 - A1.2.2. Create function in Java which calls sql function to check if username exists
- A1.3 As a user I want to ensure my password is between 8 and 32 characters long so that my account is more secure
 - A1.3.1. Add constraints to database
 - A1.3.1. Create Java function with constraints
- A1.4 As a user I want to ensure my username is between 6 and 30 characters long to reduce the likelihood of others trying to use the same username (3 points)
 - A1.4.1. Add username constraints to database
 - A1.4.1. Create function with username constraints
- A1.5 As a customer I want to be able to enter my name so that it will appear on my account (5 points)
 - A1.5.1. Create/add to a Java function to take in a first name and last name from the user during registration
 - A1.5.2. Create/add to a sql procedure to store first name and last name in the database
- User story E1.3 As a user I want to be able to enter my home address so that it can be on my bookings
 - E1.3.1. User enters address that exists in the database. Function/procedure required to add new address details to the address table

- E1.3.2. User enters a first line of address and postcode that doesn't exist in the database - functions required to take in city and country from user
- E1.3.3. User enters a city that doesn't exist in the database - function/procedure required to add new city to the database
- E1.3.4. User enters a country that doesn't exist in the database - function/procedure required to add new country to the database
- E1.3.5. After user enters address details, function/procedure required to return address_id and add it to user_account table

Before writing the code for the user stories, I created a flow diagram on Miro (figure 9) to help visualise the user's journey through the process of creating an account and/or logging in. The pink box denotes the point at which my initial register/login menu connects with Agnes' main menu.

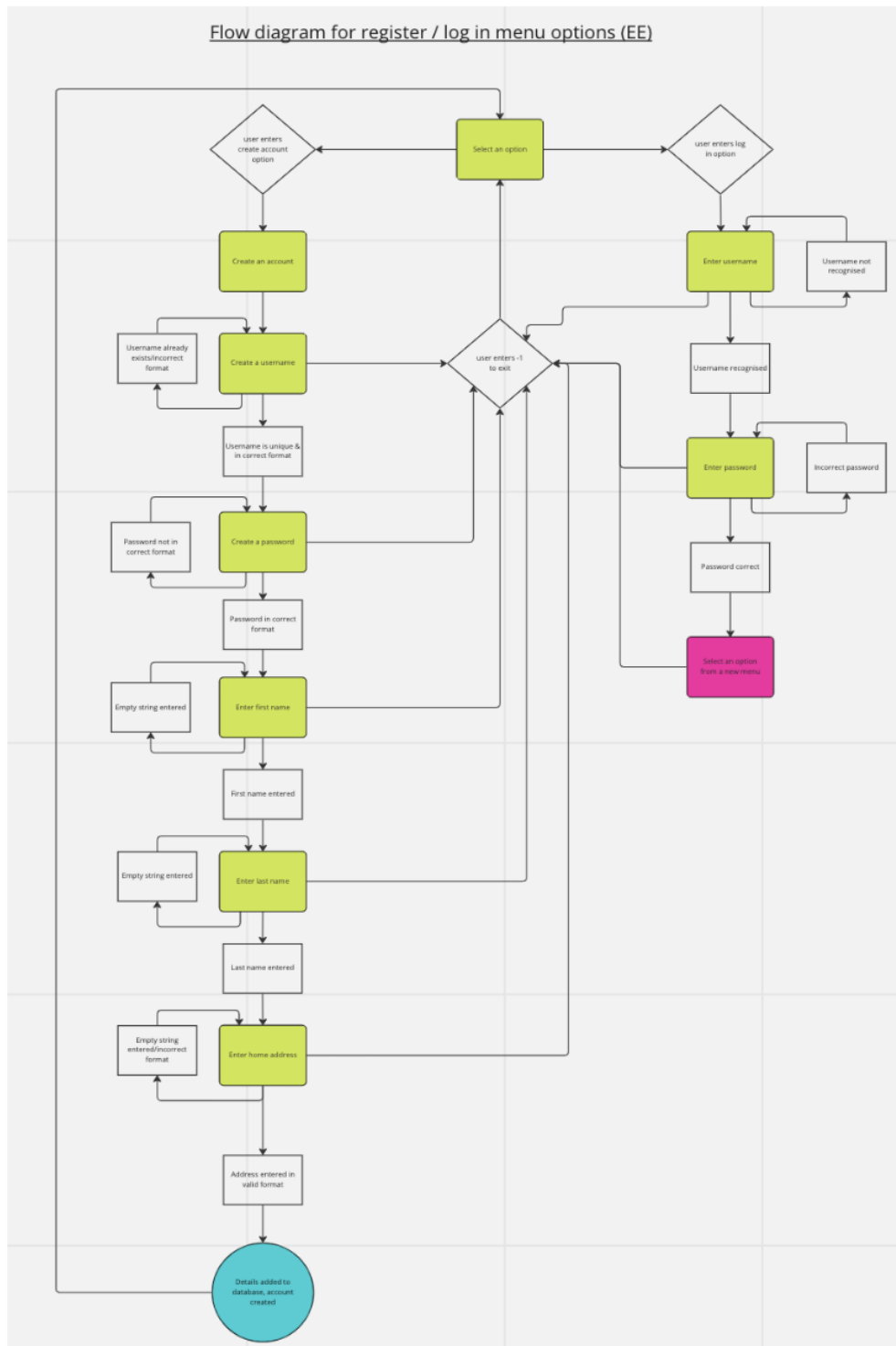


Figure 9: Flow Diagram Demonstrating the Process for Creating an Account / Logging in

PostgreSQL

We created all database tables during our group session. This included a 'user_account' table with the following attributes:

Attribute	Characteristics/constraints	Reasoning for constraints
username	Primary key, not null, varchar, 6-30 characters	Usernames are unique therefore good candidate for a primary key, usernames are conventionally around 6-30 characters long
first_name	Not null, varchar, up to 255 characters	First names can be very long
last_name	Not null, varchar, up to 255 characters	Last names can be very long
password	Not null, varchar, 8-32 characters	Passwords are conventionally around 8-32 characters long
address_id	Foreign key, not null, integer ¹	Set as a foreign key as it references address table & avoids repetition. No character limit as address list can increase exponentially

¹In hindsight, this should not have been set to an integer. A primary key should not be subjected to mathematical operations, therefore this should have been set to a varchar.

An 'address' table with the following attributes:

Attribute	Characteristics/constraints	Reasoning for constraints
address_id	Primary key, not null, integer	Uniquely identifies addresses
street	Not null, varchar, up to 255 characters	Street names can be very long
city_id	Foreign key, not null, integer ¹	Set as a foreign key as it references city table & avoids repetition. No character limit as city list can increase exponentially
country_id	Foreign key, not null, integer ¹	Set as a foreign key as it references country table & avoids repetition. No character limit as country list can increase exponentially

postcode	Not null, varchar, up to 7 characters	Set as a foreign key to avoid repetition, up to 7 characters as postcodes are always <=7 characters (without spaces)
----------	---------------------------------------	--

¹In hindsight, this should not have been set to an integer. A primary key should not be subjected to mathematical operations, therefore this should have been set to a varchar.

A 'city' table with the following attributes:

Attribute	Characteristics/constraints	Reasoning for constraints
city_id	Primary key, not null, integer	Uniquely identifies different cities
city	Not null, varchar, up to 50 characters	Longest city name is Winchester-on-the-Severn in Maryland which is 21 characters long. Went with 50 characters just to be safe

A 'country' table with the following attributes:

Attribute	Characteristics/constraints	Reasoning for constraints
country_id	Primary key, not null, integer	Uniquely identifies different countries
country	Not null, varchar, up to 16 characters	Longest country name in UK is 16 characters (Northern Ireland) ¹

¹In hindsight, this should not have been constrained to 16 characters. The logic behind this constraint was that the application was UK-based and countries outside the UK would not need to be stored in the database, however we didn't consider that someone's home address might not be in the UK.

I created a procedure called 'add_user_to_user_account_table' to add a new user to the user_account table. It takes in a username, password, first name, last name and address id, which would be entered by the user on the Java side. I decided to write my procedures in an sql file instead of writing them out in the Java console as it's more structured and easier to read. I also anticipated that my Java files would get quite big and wanted to make sure they were easy to read.

I created a function called 'check_if_user_account_exists' to check if a user account already exists. It takes in a username, which is entered by the user on the Java side. It checks if that username is listed in the user_account table and returns a table with the username if it exists. If it doesn't exist, an empty table is returned.

I created a function called 'check_if_address_exists' to check if an address entered by the user already exists in the database. It takes in the first line of address and postcode from the

Java console, searches the address table, and returns a table with the address id, first line of address, city, country and postcode if it exists. If it doesn't exist, no values will be returned in the table.

I also created similar functions called 'check_if_city_exists' and 'check_if_country_exists' which worked in the same way as the functions mentioned above, but for city and country.

I wrote a manual test for each procedure/function in the sql file to make sure they worked before wiring them up to the Java side (can be seen beneath each procedure/function as comments titled 'test:'). I deliberately gave my procedures and functions descriptive titles so that their purpose and functionality was clear.

I wrote some of the simpler sql statements on the Java side, e.g. inserting a single value into a table that doesn't have any other dependencies.

Java

I created a method called 'createUserAccount' which gets called when the user enters option 1 from the main menu. It takes in a username from the console, then the 'isAValidUsernameFormat' function is called to check if the username is in the correct format. It prevents the entry of a null value and ensures that the username is within 6-30 characters long, as specified in the database. Having controls on both the Java and the database sides gives us confidence that the username format will always be correct. If it is not in the correct format, the program loops until a valid entry is made. If a valid entry is made, a check is made to see if the username already exists in the database. This check is done via the 'checkIfUserAccountExists' function which calls the 'check_if_user_account_exists' function in the database and returns a result set. If the username already exists, a populated result set will be returned and the user will be prompted to enter a different username. The program will loop until the user enters a valid username or, alternatively, they are given the option to return to the main menu by entering '-1'. I decided to include the option to return to the main menu at every stage in case the user accidentally selects the wrong option.

If the username doesn't exist, an empty result set will be returned, and the user will be prompted to enter a password. The 'isAValidPasswordFormat' function is called, to check if the password is in the correct format (similar to the username validation check). Again, the program will loop until a valid entry is made.

Once a valid password has been entered, the user is asked to enter their first name, followed by their last name. The input is immediately processed by the 'capitaliseInputStringAndRemoveWhitespace' function, which gets rid of additional spaces that may have been typed in by the user, capitalizes the first letter of each name, ensures all other letters are lower case and prevents the entry of an empty string. It was important to trim unintentional spaces so that if they wanted to in future, the administrator would be able to search the database for a specific name (which would only return a match if the spacing was identical). The 'isAValidInputStringLength255' function is then called, which

checks that the corrected input string is not a null value or an empty string and does not exceed 255 characters. The program loops until a valid entry is made.

When constructing the original create account feature, I deliberately hardcoded the 'addressId' attribute within the code to match an address that pre-existed in the database. This was required because the 'address_id' attribute within the 'user_account' table was set to not null (as it was a foreign key which referenced the primary key in the 'address' table). Hardcoding this value enabled me to work on the 'create user account' feature, without having to rely on user story E1.3 being completed first. Because address_id in the user_account table was referencing the address table, and this had interdependencies with both the city and country tables, I knew that this user story was going to take a significant amount of time for me to complete (hence the 13 points assigned to it).

Once I got the 'createUserAccount' method working for the username, password, first name and last name sections, I started working on user story E1.3. I began by removing the hardcoded value from my 'addressId' variable, so that I could make this dynamically assignable to either a new address id (for an address not currently in the database) or an existing address id from the database.

After entering a password, the user is prompted to enter their first line of address. When creating the input prompt, I acknowledged that the user could potentially enter characters that violate the constraints set up in the 'address' table. To handle this, I ensured the user input was sanitized by calling the 'isValidInputStringLength255' function mentioned above. The 'capitaliseInputStringAndRemoveWhitespace' function mentioned above is also called.

The user is then prompted to enter their postcode. In a similar way, the string is formatted so that spaces are removed and all letters are changed to upper case. It is then checked by the 'isValidPostcodeFormat' function. Once the potential for bad data had been handled, I proceeded to wire up the SQL procedures I had written to insert data into the user_account, address, city and country tables.

First of all, I worked on the scenario where an address already existed within the database. I utilized the function 'check_if_address_exists', which takes in the first line of address and postcode entered by the user and returns the address if it exists. One issue I encountered was that when a user entered an existing first line of address in lower case or included additional spaces, the function did not return a result, despite it existing within the database. Because the query required an exact match, the address was not recognized. This was the reason I created the 'capitaliseInputStringAndRemoveWhitespace' function.

If the address is found, the function returns the address_id which gets assigned to the addressId variable. The value assigned to the addressId variable is then added to the user_account table alongside the username, password, first name and last name by calling the 'add_user_to_user_account_table' procedure.

The more difficult scenario was when a new address had to be added to the database. To implement this, I needed the user to input their city and country of residence. With these values, I set up the program to run the city and country functions (equivalent to the address

search) to check if there were matches in the database. If there was a match, the city/country id would be returned.

After the user enters their country of residence, the final part of the createUserAccount method calls the 'addNewAddressToDatabase' method which inserts all the returned address details (firstLineOfAddress, cityId, countryId & postcode) into the address table. This populates the address table, then the 'addressIdWithinResult' function returns a new addressId. This addressId then gets added to the user_account table as described above.

When writing the 'Prepared statements' to wire up my SQL procedures, the main challenge was figuring out the order in which the data had to be entered into the tables. Due to the dependencies established between the tables, I was unable to insert data into the address table until the corresponding city and country tables were populated and the 'city_id' and 'country_id' primary key values could be accessed. I also had to handle the scenario where the user enters a country which exists but a city that doesn't (and vice versa). I ensured that both values would be inserted into the city and country tables before executing the insert statements to add values to the address table, which had to be populated before values could be added to the user_account table.

Unit Testing

When checking that the user can only enter sanitised input into the console, I used a test driven development approach to ensure that my validation function worked as expected. This involved creating a function called 'testIsValidInputStringFormat()', which takes a string (such as the first line of address, first name or last name) and returns true or false if it has met the validation rule. I placed this function into the 'UserInputs' class inside the helpers package so that others could use it. I then created a corresponding 'UserInputsTest' class which has within it a 'testIsValidInputStringFormat' function that runs a variety of possible inputs (some of which utilized the 'isValidInputStringFormat255' function within it). This test checks that the string passed into the 'isValidInputStringFormat255' function returns the expected boolean output. For example, if an empty string was passed into the function, then the expected output is false as it is not a valid format. I made use of boundary value testing, checking the boundaries of what was considered 'valid'. The boundaries were based on the database constraints e.g. a first name or first line of address had to be between 0 and 255 characters long. I also checked against other common causes of bad data such as empty strings and white spaces.

An additional test was added to check the behaviour of the corrective function 'testCapitalizeInputStringAndRemoveWhitespace()'. To test this I wrote a function called 'testCapitalizeInputStringAndRemoveWhitespace()' and passed in empty strings, strings with white spaces and strings which were entered in lowercase.

I also wrote the test 'testIsValidPostcodeFormat()' to check that the function 'testIsValidPostcodeFormat()' returns the correct boolean value when the postcode format is entered.

Summary of Tasks I Completed

I worked on the following features specified within the coursework brief:

- Register to gain a username and password
- Login using the name and password
- Enter name and home address

The user stories assigned to and completed by me were:

- A1.1 As a customer I want to create an account so that I can securely interact with the application (8 points)
- A1.2 As a customer I want the application to prevent duplication of usernames so that I can be uniquely identified (8 points)
- A1.3 As a user I want to ensure my password is between 8 and 32 characters long so that my account is more secure (5 points)
- A1.4 As a user I want to ensure my username is between 6 and 30 characters long to reduce the likelihood of others trying to use the same username (3 points)
- A1.5 As a customer I want to be able to enter my name so that it will appear on my account (5 points)
- E1.3 As a user I want to be able to enter my home address so that it can be on my bookings (13 points)
- B1.1 As a user I want to be able to input my username and password so that I can log into my account (8 points)
- B1.2 As a user, after I have logged in I want to be able to view options available to me so that I can make bookings (13 points)
- B1.3 As a user I want to be able to go back to the login menu so that I can choose different booking options (5 points)

Everything I've written in the code base has my initials (EE) either in the file name and/or the comments section.

I created everything in the following files:

- EE-register-or-log-into-account.sql
- RegisterAndLogin.java
- RegisterAndLoginTest.java

And I made some contributions to the following files:

- Main.java
- MainTest.java
- Database.java
- DatabaseTest.java
- UserInputs.java
- UserInputsTest.java

Total number of user story points I completed: 57 out of the 217 points in the sprint backlog (26% of the sprint backlog). The total number of story points actually completed by the team during the sprint was 197 (C3.2. was not completed), so my relative contribution to the completed was 29%. I felt that everyone contributed equally towards the project and we all worked well as a team.

Reflection

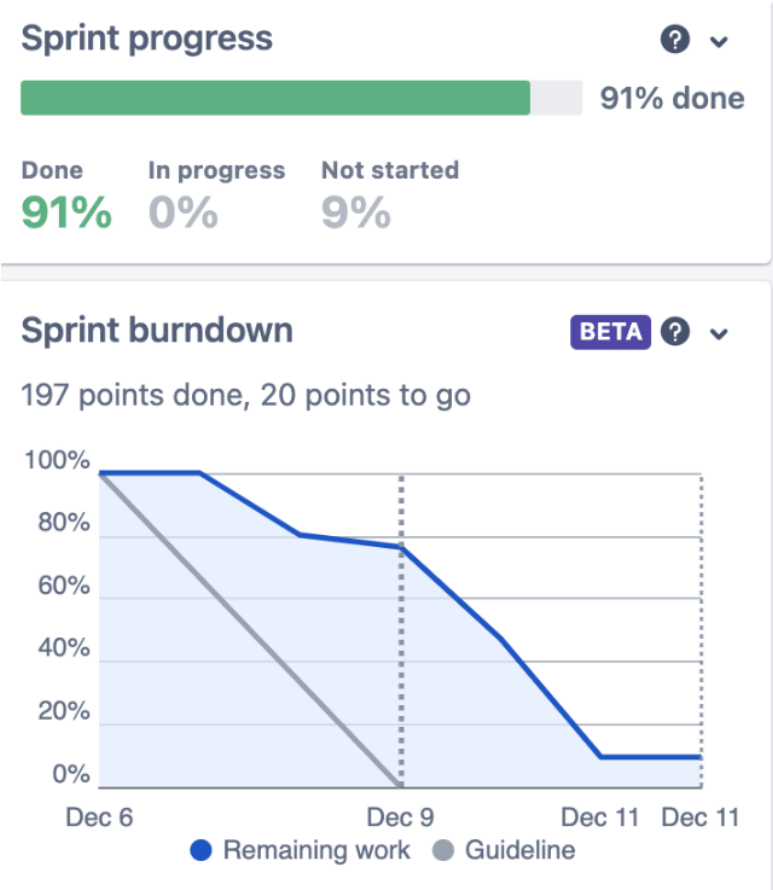


Figure 10: Burndown Chart

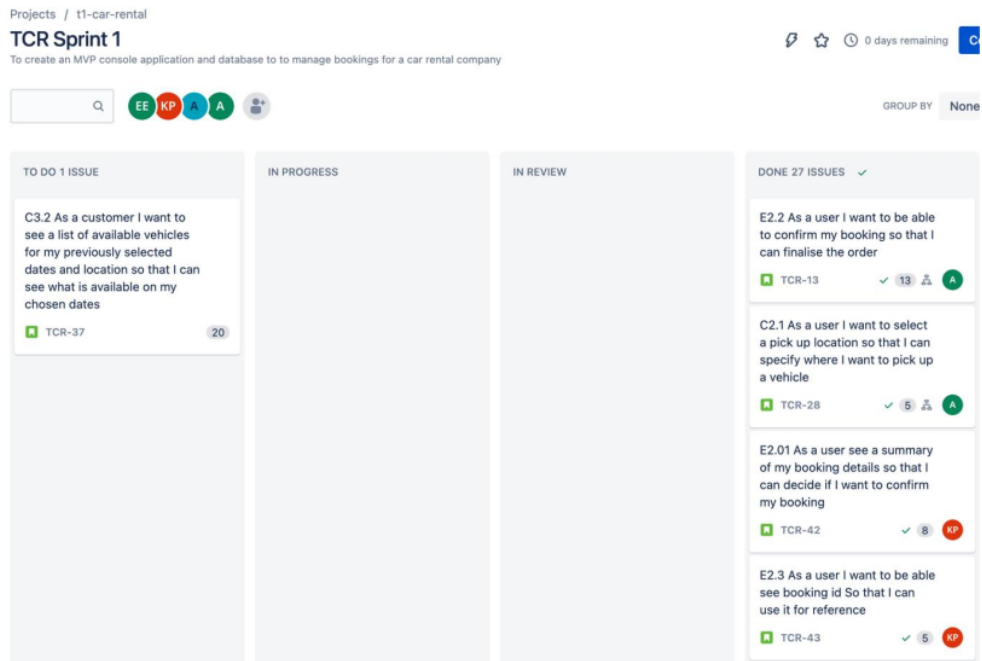


Figure 11: Screenshot of Jira Board after Completing the Sprint

Overall, I was happy with the end product as we were able to deliver a functional MVP in an extremely short amount of time. We worked together well as a team and everyone contributed equally towards the project. Everyone made frequent commits to Git and performed thorough reviews on other members' code. Each team member had different strengths, for example, Arthur demonstrated great knowledge of postgres, Agnes was very organised and ensured that we met up regularly and Kai was good at identifying issues in the codebase.

The Agile process invoked a lot of valuable discussions between team members and the daily meetings facilitated the identification and subsequent resolution of blockers. Our daily stand-ups ensured that we were all up to date with each other's progress and helped to keep everyone accountable and organised. The sprint lasted 4.5 days, 2 of which were spent planning how we were going to approach the project as a team, discussing the overall design of the MVP, estimating and assigning user stories. Although the planning stage took a long time, I believe that it played a crucial part in the delivery of a successful end product. Had we not spent this time planning, we likely would have encountered a lot more issues, e.g. duplication of work from a lack of communication.

The biggest flaw in our application was caused by the failure to complete user story C3.2 'As a customer I want to see a list of available vehicles for my previously selected dates and location so that I can see what is available on my chosen dates'. It was an important part of the program as the user must be able to see a list of available vehicles on their selected booking dates before they know which car registration to enter for their booking. The story could not be completed as we ran out of time. This was not surprising as we didn't have a

sprint velocity to refer to when selecting the user stories for the sprint. Despite this, we still managed to complete 91% of the sprint backlog (figure 10).

There were numerous occasions where members of the team broke the build as they pushed directly to the main branch. We had agreed as a team that we should be branching off main when working on our assigned user stories, however due to lack of discipline/experience this rule was broken. In addition to this, some team members merged pull requests before gaining 2 approvals, which went against the rules we'd set ourselves at the beginning of the project. This proved problematic and we had to spend a lot of time resolving compilation errors. One way of mitigating this would have been to create organisational GitHub accounts. Another option was to make the repository public, however, neither of these options were appropriate (as mentioned earlier in this report).

Had we been more experienced, it would have been beneficial to add a CI pipeline using tooling such as Drone, so that every time a commit was pushed to the main branch, Github would alert us if the build failed. Furthermore, if all the team members added a higher percentage of code coverage with unit testing, we would have been able to run the unit tests as part of the CI pipeline to identify potential logic errors/bugs. Even without implementing a CI pipeline, it would have been useful to have more unit tests to help with debugging, giving us more confidence in the functionality of our program. Apart from the mock database connection, we did not have any database unit tests in place. After creating the mock database connection, I made an attempt to create a mock result set, however I wasn't able to implement it as I found it too challenging.

If we had more time, I feel that pair programming would have been useful when people were working on adjoining features. During the final stage of the sprint, the team identified some gaps in the application relating to user story E2.2, where the payment table should have been updated once the booking was confirmed by the user, however due to miscommunication, this was interpreted as just returning a message to the console stating the customer had paid (without updating the payment table). As this user story was closely related to E2.1, which was to enter payment details, this discrepancy would likely have been identified had both members engaged in pair programming. This wasn't noticed until the coding cut-off point that we'd agreed on.

It was slightly inconvenient that each team member was running their Postgres database on different ports. The standard port number is '5432', however when merges were occurring some team members were setting theirs to '5433' or '15333'. Half way through the project, I came up with a solution which involved adding the port number to the db.properties file. This meant that the port number was configurable for each team member so that when pushes were made to Git, it would not impact other team members when they ran the code locally. In hindsight, I believe it would have been better to produce a docker-compose file that could be used by every team member. This would have enabled us to work on the same version of Postgres in the same local environment, without having configuration conflicts.

Lastly, an issue I found towards the end of the sprint was the inconsistency in formatting of the menu options displayed on the console. Some menu options were delimited with brackets, whilst others used colons. This was the result of isolated feature development and

lack of pair programming, as every member created their own menu system with limited discussion.

Conclusion

Over the period of the 5-day sprint, the team collectively built a car rental company application that allows a customer to:

- Register for an account
- Log into their account
- View and select pickup and drop off locations
- Browse available vehicles at pickup location
- Search for a car by make/model at selected pickup location
- Select a rental vehicle to view its availability
- Enter the dates that they want to make a rental on
- Create a new booking
- View and finalise their booking

Figure 12 visualises all the different menu options available and demonstrates the relationships between each feature.

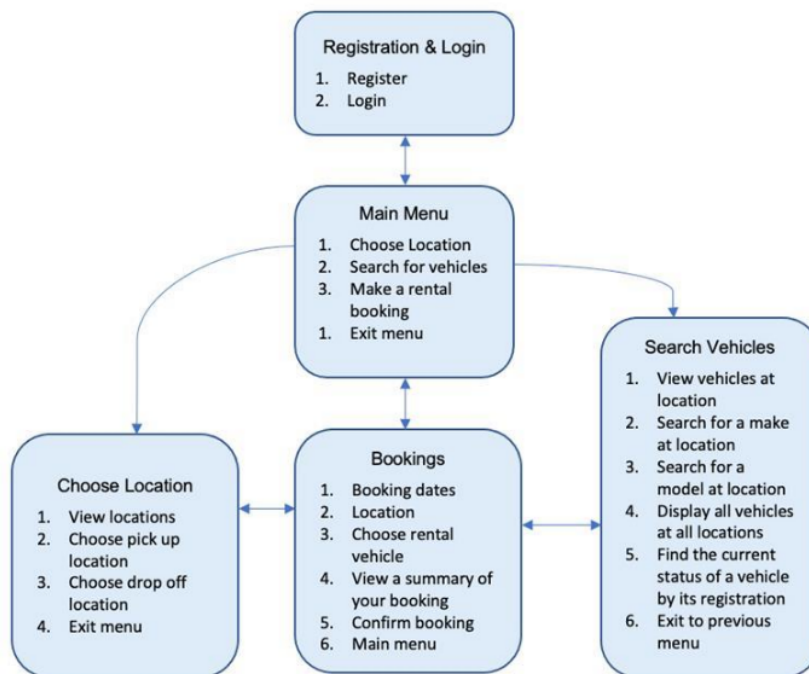


Figure 12: Flow Diagram of Different Menus Within the Program (created by Agnes)

FINAL GRADE

79/100

GENERAL COMMENTS

Instructor

Sw Des Eval (30): 24
Database Eval (30): 22
Code & Test (30): 23
Reflection (10): 10
Feedback:
=====

Good team work and software design. The evaluation is good but it could have been a more coherent narrative.

Some good work in contributing to the software development agenda of the team. A well-written report.

The database design looks a little strange and parts of it are not in 3NF. Evidence of teamwork in design, but more evaluation of the design choices could have been provided.

PAGE 1

PAGE 2

PAGE 3

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

PAGE 22

PAGE 23
