

blockAssignment_OJordan_P14 4558

by Oliver Jordan

Submission date: 11-Dec-2022 10:54AM (UTC+0000)

Submission ID: 191846036

File name: 2911_Oliver_Jordan_blockAssignment_OJordan_P144558_472250_355794382.docx (1,013.47K)

Word count: 2295

Character count: 11011

Assignment 2: Block Test Evaluation and Reflection

Oliver Jordan (P144558)

DVLA: MSc Software Engineering (Web Development)

ACCA7000/ACCA7001/ACCA7002

Mike Dacey, Tim Bashford, Stephen Hole

Contents

1	Introduction	3
2	Evaluation of the Design	3
2.1	Project Planning	3
2.2	Building the Database	4
2.3	Building the Java Code	6
2.3.1	Searching Bookings with Terms.....	6
2.3.2	Changing the Booking Details	7
2.3.3	Checking a vehicle's bookings.....	7
2.4	Writing unit tests.....	7
3	Evaluation of the project as a whole	7
4	Conclusion.....	8
5	Appendices.....	8
5.1	Resources	8
5.2	Database Connection Details	8

1 Introduction

As part of the assessment of some of the modules undertaken on this course, a group project was completed to demonstrate the skills taught across three of them. This report aims to, as part of said assessment, evaluate and reflect upon that process. Links to relevant resources can be found in the appendices.

2 Evaluation of the Design

2.1 Project Planning

The brief for this project tasked us to design a database system for a car rental company, with user inputs being taken through a console-based java application. The team decided to plot out a user story map first of all, in order to ensure that a clear understanding of the tasks was shared throughout us all. With task listed and broken down, we were able to plot out which release each feature should appear in and produced a list of stories aimed at producing a minimum viable product.



Figure 1: The User Story Map the team created

These stories were then placed into Jira, a tool useful for managing agile development, where we could place them in a sprint backlog and use a Kanban board to track progress. With our stories on hand, the team then estimated using planning poker, ensuring that each task had a story point estimate that reflected the effort needed for a task while considering the team as a whole.

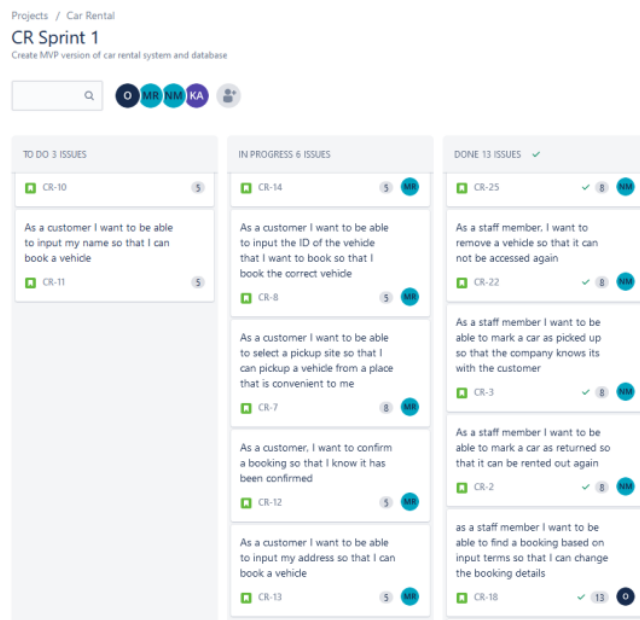


Figure 2: A sample of the Jira Kanban board

This initial planning stage helped ensure that the scope of this iteration of the product remained manageable and helped prevent me from going off track. It also ensured that tasks I was working on would not overlap with others unless we had specifically planned to. The estimation ensured that I understood the size of the tasks I was taking on, and that difficulty of it was a reflection of the team's perception. This meant that while certain larger point stories may not be as difficult in my eyes, I could understand that taking them on would clear a lot of effort from the team as a whole.

2.2 Building the Database

Before any stories could be taken on, we decided to draw up a design of the database, given that we knew what data fields we would need for our MVP. This would ensure that we all were working from the same data design and prevent sudden rewrites of the code as members came up with their own designs and put code together. However, partway through the project, it was discovered that a team member had been working on an altered design of the database – this was handled with a few quick meetings with team and evaluating what code would need to be refactored should we take on this new design. Given that it was a substantial amount and the project timeframe was relatively short, it was decided to compromise and revert the parts of the database already being accessed in code so that we could continue to progress. Had there been more time with this project, it would've been likely we took on this new database design, and in order to allow this to still be shown in the project, as many aspects of the new design were kept as possible.

The initial design was drafted collaboratively, with every team member being present and providing input. Once this design was done, we spread out the work of creating tables between us all and once completed, added our various constraints.

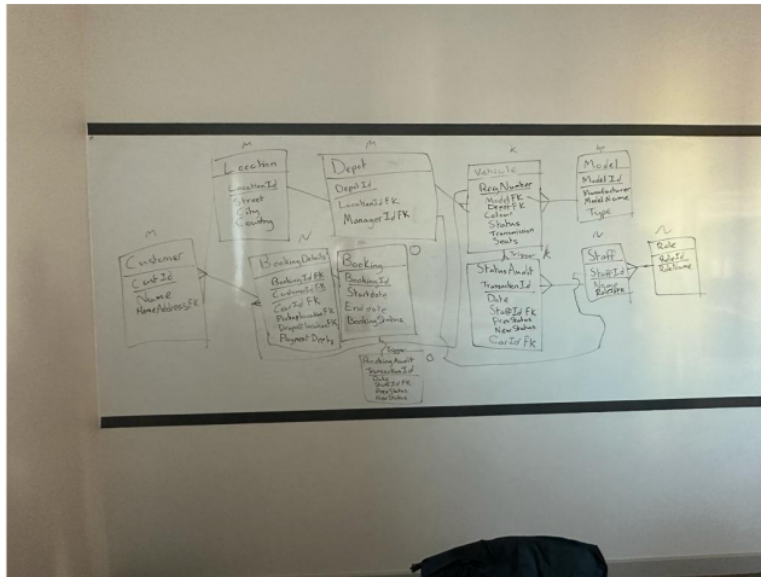


Figure 3: The initial database ERD plan. Foreign keys are marked with FK, while primary keys are underlined.

The final database design makes use of separate schemas to separate the audit tables being used from the rest of the database. The public schema contains the information about cars, depots and bookings, along with some other needed data. This design works well, but holds some small areas which may be improved – for example, one of the later releases requires a log in system to be created, which could benefit from the addition of roles to the database. The audit tables do not contain foreign keys within them – this may have just been an oversight during their design and will make understanding the full data it is supposed to reference hard to collate as joins may not work on the data within. These audit tables make a note whenever there is a change to certain tables, through the use of trigger functions, so that any unauthorised or mistaken changes can be checked.

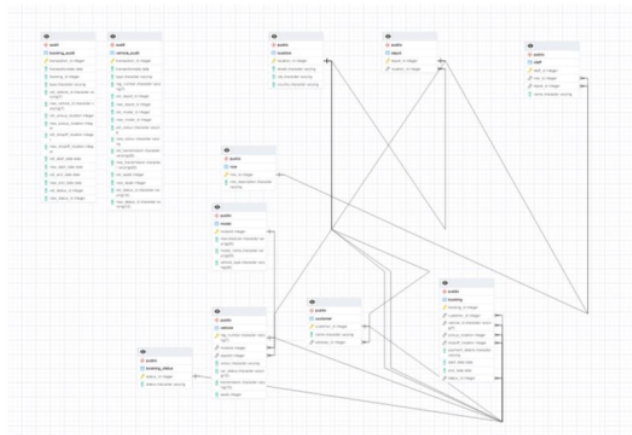


Figure 4: The ERD of the final database design, generated through Postgres

2.3 Building the Java Code

With a database design in place, the team could set off and start coding their respective parts of the java frontend. I programmed some menu objects that could execute a given function when they were selected by a user to cut down on the amount of switch statements that were needed, allowing the team to write their functions without worrying too much about how it would fit into a menu. I did later make use of the same structures within some functions as I quite like the code, but the others have used switch cases for additional submenus within their code where needed (I believe Mihai has made use of my objects in his own code however).

The program is also built so that a set of SQL scripts are read in from a file and sent to the database in order to ensure the data is in the same state every time for demoing and testing purposes. The main program and unit tests do, however, point at separate databases on the server, ready for when the main database is used to hold data properly. The scripts for rebuilding were written mainly by Mihai, with some edits by myself in order to prevent code reformatting and to allow it to be read with ease into java code.

I picked up three separate stories from our selection:

- as a staff member I want to be able to find a booking based on input terms so that I can change the booking details.
- as a staff member I want to be able to mark a booking as cancelled so that a vehicle may become free for use again.
- As a staff member, I want to check the bookings on the vehicles so that there is no conflicting time.

These totalled 34 story points, which falls close to the totals the other members completed. The following sections will evaluate the code written for each story.

2.3.1 Searching Bookings with Terms

This code takes input for a customer name, a vehicle registration and a status ID, then searches the existing bookings for matches. If the user does not type anything in a field, then that is omitted from the filter. Typing nothing as a result, returns all bookings. Input for the status ID must be in range of the existing status numbers.

The code will build two halves of a statement – the first half is the same every time and selects records from the booking table then creates some joins on relevant tables. The second half is built using the user inputs, which are stored in a class in code for neatness. This second half only begins to build if an input is found in the field it is currently checking, so that if nothing is found, nothing is appended when the full query is built, creating the behaviour explained above.

This code tackles the complexity of having to append WHERE statements at the right time, as if they are appended and left blank, an incorrect statement will be used, limiting the usability of JSQL's prepared statement parameters (PreparedStatement.setString ect.) here. There is space for improvement with regards to user validation – registration plate input can take any length, which would result in no results being returned. There is no ordering on the returned result set, and could be done in the SQL query as per the users request, perhaps by prompting for it before the string finishes building.

2.3.2 Changing the Booking Details

For this task, code needed to be written that would change the status of an existing booking to something else. In order to do this, user input was first requested, asking for the booking ID and the status. In order to allow a user to see the current bookings available, an option to show all bookings is also provided up front. While the status id was validated to be within a range, as the statuses that a booking can have are limited not expected to change, the booking ID input is not and will print an error should the user input an ID that does not exist.

With inputs in hand, the code then builds an SQL statement making use of the UPDATE statement and executes it. Due to a trigger being present on the booking table in the database, this will also result in a new entry in the booking_audit table under the audit schema.

This code functions correctly, however there are some areas it could improve upon. Firstly, a more elegant way to handle inputting an ID that does not exist is certainly possible. It may also be beneficial to write code that allows you to directly select a record to update from a list you are shown, as it may be more user friendly. The error handling here, and throughout my code, simply prints the error code – perhaps some code logic would've been useful here to execute on an error.

2.3.3 Checking a vehicle's bookings

This code will search the bookings table for any bookings containing an inputted registration plate. This was on the simpler side to write, as it was one input feeding into an SQL statement that would be identical each time. For this code, I could use prepared statement parameters and feed in the inputted string. While this story does overlap with the one described in section 2.3.1, the output differs as this story was created to allow a user to check that a vehicle had no conflicting time. In a further release this could be done manually when a customer creates a booking, ensuring that dates for a vehicle never overlap.

2.4 Writing unit tests

I have written unit tests relating to the stories I have worked on, testing functions that change data on the database by retrieving data after the operation to see if the expected change is there. The program is set up in such a way that the data on the database always starts in the same state to facilitate this. Functions that retrieve data based on user input were easy to test, simply requiring said same state to allow for predictability when retrieving results. In the case of testing the function that changes booking status, an additional test is also performed to ensure that the booking_audit table has had data input reflecting the change.

3 Evaluation of the project as a whole

Given the short timeframe for this project, I think the team has done well to create a minimum viable product version of the brief. While the system lacks a login system as of this version, the product should be able to be used by a staff member, making bookings on behalf of a customer so that staff features are not accessible to customers.

Overall the team worked well together – we stuck to our stories, adapted quickly to complications and kept communication up throughout the sprint. Although we did not make use of a scrum master or a daily stand-up ceremony, the majority of the team worked in the same room together, enabling constant communication to flow, and I made sure to contact the team member not on site with us as appropriate to allow for us to all stay up to date with each other. I believe the team has undertaken a similar level of work when regarding the stories that were chosen, and nobody seemed to feel the need to voice that they had stories they did not want or could not complete.

The team made use of a GitHub repository to collaboratively build a codebase. This was set up with protection on the main branch so that at least one other team member had to review the code before it could be merged into the main branch. This allowed us to have a smooth experience when putting our various features together throughout the project and helped ensure time was spent creating new code rather than fixing old code.

4 Conclusion

In conclusion, this project has been an interesting exercise in collaborative programming. I believe the team has worked well together and produced a suitable minimum viable product for the timeframe provided, making use of relevant agile processes to ensure this outcome.

5 Appendices

5.1 Resources

Miro Board containing User Story Map: https://miro.com/app/board/uXjVP8q41cQ=

Jira Board: <https://t2unicarproject.atlassian.net/jira/software/projects/CR/boards/2>

GitHub Repo: <https://github.com/glugg12/UniCarRentalProject>

Most recent commit at time of writing:

<https://github.com/glugg12/UniCarRentalProject/commit/553996749be294f677a0ef999f2b19a98c1718e8>

Code Demo: <https://www.youtube.com/watch?v=gxOKW12eIPc>

5.2 Database Connection Details

To access the remote database, use the following details:

- IP: 139.162.251.67
- Port: 5432
- User: postgres
- Password: admin

The databases within the remote that the code accesses are CarRentalMain for the standard running of the project and junitCloneData for the test functions.

FINAL GRADE

70/100

GENERAL COMMENTS

Instructor

Sw Des Eval (30): 21
Database Eval (30): 23
Code & Test (30): 21
Reflection (10): 5
Feedback:
=====

Good team design but the evaluation could have provided a little more detail on story estimation and sprint planning.

Good database design with evidence of team work, evaluation was a bit brief.

We were expecting a little more reflecton on the challenges you faced.

PAGE 1

PAGE 2

PAGE 3

PAGE 4

PAGE 5

PAGE 6

PAGE 7

PAGE 8