

Controlled Assessment: Report

Bradley Watkins

December 2024

Contents

Abstract

Within this report will be a discussion and evaluation of the layered software design chosen as a basis for the Alu-car-d vehicle rentals service, the database implemented part of that service and the agile principles that drove the working practices leading to the creation of the service.

1 Software Design Evaluation

When designing our console application, the key points to consider were testability, collaboration and everyone's confidence level. It was this final point, the confidence level of those in the team, that was the driving force, however. I suggested we use what we had learnt during our time at the DVLA and aim for a layered design that would allow for greater separation of concerns and abstracts away unnecessary detail when working on different parts of the system.

The layered design we used can be broken down into three key layers, where each layer receives its necessary dependencies via dependency injection.

These layers are:

1.1 Repository Layer

The repository layer deals with data manipulation and database integration. Here database queries will be made via SQL to create, read, and update data in our database. When retrieving data from the database via the repository layer, data and models should be appropriate and closely resemble the models found in the database.

Making use of the repository layer made it much easier to work with the database, as there was a good understanding of where to find existing database integration code, and where any new code that interacts with the database needed to be created. The connection to the database should be passed to this layer via dependency injection.

1.2 Service Layer

The service layer is concerned with handling business logic, such as data validation, parsing, and filtering. This layer handles incoming data, which may be transformed before being passed to the repository layer, as well as outgoing data from the repository layer, which may be transformed before being returned to the caller.

This layer should receive any necessary repository objects via dependency injection, allowing them to be mocked and for this layer to be easily tested. This also allows the service layer to abstract away database integration logic from any higher layers, which no longer need an understanding of what form the data takes or what method is used to persist the data.

1.3 User Interaction Layer

The user interaction layer deals with application flow, providing the user with context via output to the console and taking relevant input from the user to achieve a desired goal. This is the topmost layer, receiving any relevant services required to achieve a goal, such as making a booking or logging in, via dependency injection. This layer is created and called in the main method.

1.4 Evaluation

The layered software design pattern helped in many ways but also caused some confusion and loss of velocity in others. In general it was a good decision, mapping well to the design we used at the DVLA and making testing relatively simple via mocking with Mockito. Collaboration was simple, as we could all work on similar features, solving the problem in different layers.

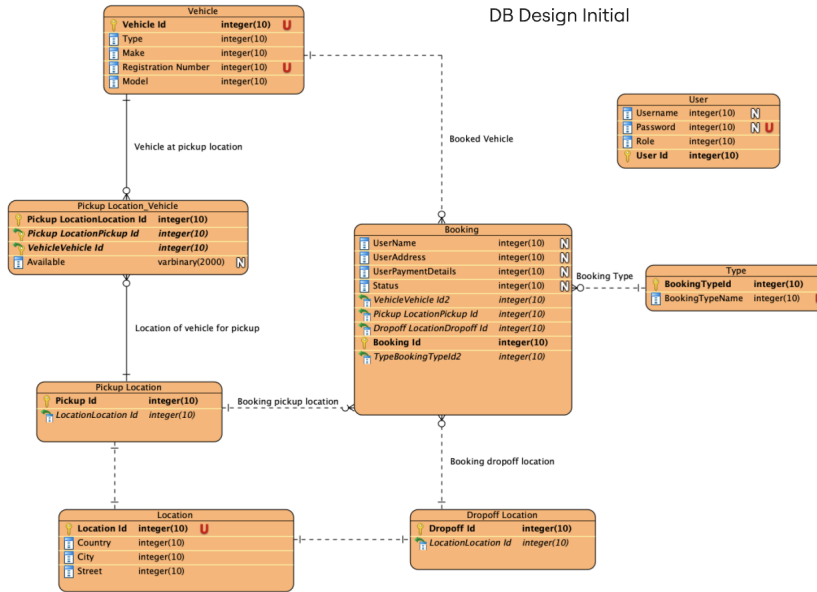
Where it slowed us down was in the initial setup and in determining how best to solve each problem and in what layer certain logic should reside. If we had simplified the design, we may have been able to accelerate the work completed. This likely would have harmed testability and code confidence overall, though, so I am generally pleased with our decision here.

Beyond the layered architecture we made use of other software design patterns, such as the factory pattern and singleton pattern, to make using the relevant parts of our system as simple as possible. This abstracted

away some of the complexity of creating any given layer, making integrating each other's code easier.

2 Database Evaluation and Implementation

The design of the database was a collaborative effort, each of us presenting ideas for tables and relationships and then iterating on these until we had our initial design. This was further modified and iterated upon as tasks were completed and flaws in our initial logic were identified. We each handled writing the SQL scripts for table definitions of separate tables to speed up the process. Later I took on the role of implementing further necessary changes to the database as they arose, as well as implementing table and data generation from code when running the application.



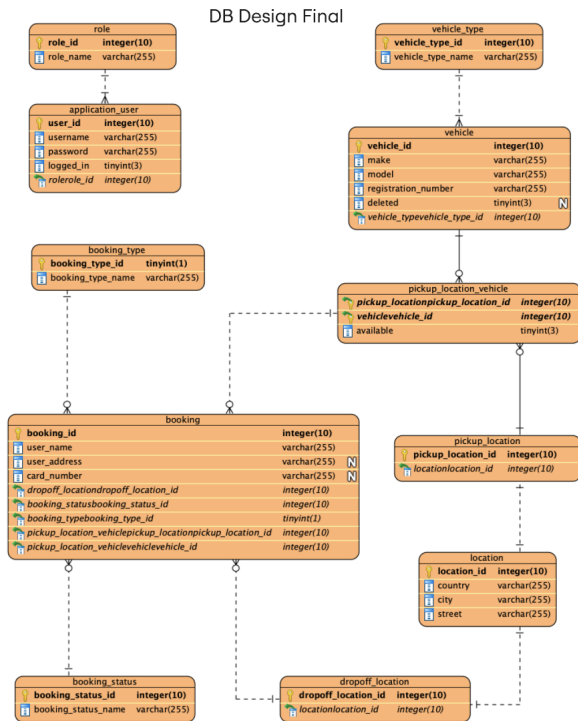
Our initial design of the database, as seen above, held up well, and the upfront time spent on this paid off. Although changes were made, there were no major overhauls, and generally we avoided confusion. From the brief, we determined our core tables would be vehicles, location, booking, and user. From these we determined that pickup locations and dropoff locations should be child instances of location, denoting their slightly varied behaviour. pickup location and vehicle then required a join table to indicate at what locations a vehicle can be found and its availability in relation to bookings.

A key decision made early on in the design process was that rentals, maintenance, and repairs should all be types of bookings that would modify the behaviour of the application in subtle ways. A booking would begin as type booking and later toggle to a rental when a pickup had been recorded. This meant all booking operations

would be applicable to all booking types, avoiding duplication in logic to achieve very similar outcomes.

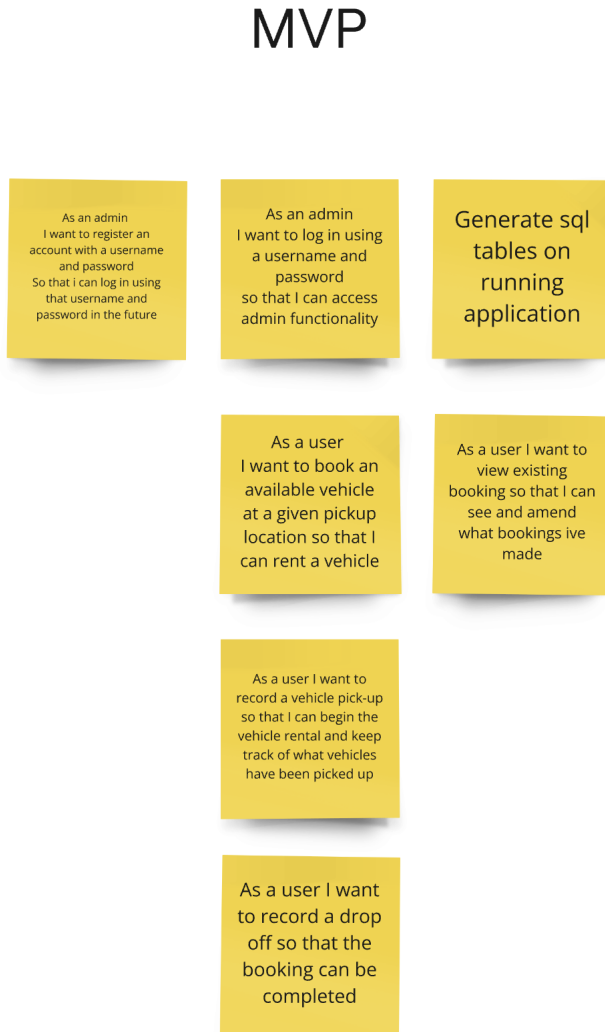
One flaw in logic that we discovered was that of deleting vehicle records when the user wanted to sell a vehicle. Due to the nested referential constraints, deleting a vehicle record would mean deleting a historical booking. This was behaviour that we wished to avoid, so we had to find a way around this. Allowing hanging references was considered; however, we felt this was a messy solution to the problem. We settled on the creation of a deleted column in the vehicles table, which, when true, would denote that the vehicle should not be considered for any operation within the application and should otherwise be treated as if it did not exist. This allowed us to retain all historical records and keep the design of the database clean. This did introduce some further complexity to the query design within the application and introduced a bug where a vehicle with the same registration number as a sold vehicle could not be created.

Our final database design can be seen below. The key changes were to extract all status and type enum values into their own tables so that these could be constrained, as well as the above mentioned change to the vehicles table. On reflection, the pickup location table was unnecessary, with a one-to-many relationship between vehicles and locations representing the same relationship with two fewer tables. Although imperfect, the database design we finished with made working with our data relatively simple and helped us to achieve all of our goals.



3 Individual Contribution

Here i'll discuss three major user stories that I worked on. These were all included in our MVP, which you can see below:



3.1 Login and Register

As an admin I want to log in using a username and password so that I can access admin functionality

The first step in achieving this user story was to break it into smaller achievable subtasks.

These were to:

- Allow the user to enter a username and password, or provide a username and/ or password via program arguments
- Compare against existing users
- Log in if user exists
- Fail gracefully if user does not exist

As we opted for a layered software design, I first began work on the repository layer. Here I created a method to retrieve a user from the database by username and password using SQL. Once this was created, I moved up into our service layer and began work on the business logic of logging a user in to the application.

It was established early that no passwords should be saved as plain text in the database, so I implemented a security utility that would handle salting and hashing the plaintext password using SHA-512, which will later be compared against the passwords stored in the database. Within the service layer, the provided username and hashed password will be compared against users in the database, with the request for existing users passing through the repository layer. If the user exists, they will be logged in, and the user model will be stored in a static variable that can be accessed anywhere in the application that can be used to check the logged-in status and the user's access role. If the user does not exist, the application will fail gracefully, letting the user know that their login has failed.

As an admin I want to register an account with a username and password so that I can log in using that username and password in the future

Although technically a separate user story, as this and logging in are similar in functionality, I will discuss this here. Breaking user registration down into subtasks was similar to user login, which you can see as follows:

- Allow the user to enter a username and password
- Compare username against existing users
- If no user with that username exists, hash the provided password and store the new user in the database
- If the user already exists then fail registration gracefully

Without going into too much detail, the flow of completing this user story followed a similar pattern to that of the user login story. I began in the repository layer, setting up the SQL query for the creation of a new user and checking that unique constraints are working as expected on the username. I then moved up to the

service layer, working on the business logic and implementing checks for null values and uniqueness, as well as hashing of the password prior to a new user being saved in the database. Finally, I worked on the registration user interface that handled retrieving the username and password for registration and providing the user with context clues as to whether something has gone wrong during registration or whether their registration has been successful.

3.2 Database Migration

As a user I want database tables and migration to be completed when running the application so that I can focus on using the application with minimal setup required

This user story was an important one in relation to ease of use and created a massive improvement in relation to user experience. Prior to the completion of this story, use of the application required running various SQL scripts within PgAdmin to set up the required tables and insert our various default records.

As this story required no user input It could be broken down in to the following logical steps:

- Translating the SQL from their current scripts into SQL queries within the Java environment
- Running the relevant repository methods from our service layer in the correct order

The first step was to translate our current database definition scripts into parameterised queries that could be run in the Java environment. This meant creating a migration repository that would contain methods for each of our table creations and data insertions. Moving up from this layer to the migrations service, we have a single method `migrate` that handles calling the various repository methods to set up the database and insert our data.

This was done in a test-driven manner, with the migration repository achieving maximum test coverage where we ensure that all of the repository methods are called and in the correct order. As mentioned, there is no user interaction involved in this story, so once the service and repository layers were completed, it was a case of setting up our migration factory to handle the creation and injection of our necessary layers and dependencies and calling the `migrate` method in main.

Completion of this task massively improved our workflow and velocity, as any changes to the database design and structure could be implemented by simply deleting the existing `ALU-CAR-D` database, creating a new empty database in our PostgreSQL server, and running the application to get our most up-to-date changes. If provided more time, I would have implemented a version number that could be stored in the database, which, when less

than the application version number, would handle overwriting the current database design to remove the need for manual destruction and creation of the database node.

3.3 Make a Booking

As a user I want to book an available vehicle at a given pickup location so that I can rent a vehicle.

This story was a major step towards achieving our MVP and required functionality from each member of the team to be in place before it could realistically be achieved. A booking references a vehicle, a pickup location, and a drop-off location, all of which were features handled by different members of the team, so work on this feature required the greatest level of teamwork.

As with the other stories, this story was first broken in to it's relevant tasks which are as follows:

- Allow the user to select an existing pickup location
- Allow the user to select an available vehicle from that pickup location
- Allow the user to select a dropoff location
- Allow the user to select a booking type
- Allow the user to enter pertinent details such as name, address and payment information
- Handle parsing and creation of the booking model using the users selected details
- Insert the booking into the database with valid referential constraints in place

When comparing this list of subtasks to the other stories I had completed, it certainly appeared far more overwhelming. This overwhelm was eased, though, as much of the integration work had been handled by the team, such as retrieving all pickup and drop-off locations and retrieving available vehicles. As with the other tasks I began at the repository layer, ensuring that a valid booking could be created from within the application using static data. Here I broke some of the philosophy of our layered design due to the implementation of our booking model. The booking model contains a vehicle but does not contain the pickup location. Rather than rewrite some of the logic that had already been implemented, I opted for velocity over quality and used the repository layer to retrieve the pickup location vehicle ID from the database using the vehicle ID as a basis. This meant constructing multiple queries within a single method, breaking the single responsibility principle and making testing somewhat harder. In retrospect it would have been cleaner and easier to change up the design at this stage, but it was a valuable teaching moment regardless.

Once the repository layer was in place, I added the service layer to handle our business logic, including preventing a booking being made on a vehicle that is not available and marking the selected vehicle as unavailable when a successful booking is made. Finally, I could begin work on the user interaction layer, which required multiple service dependencies due to bookings referencing many other tables. Dependency injection allowed me to continue a test-driven approach at this stage, ensuring that the user is receiving the correct context clues and being prompted for valid input at the relevant points within the method.

It was at this stage that I saw the major benefits to a design decision made early on with input and output objects. These objects are injected into all of our user interaction layers and provided an easy way of mocking out user interaction and checking what the user would see at any given time. They also provided utility functionality such as `nextLineFromOptions` which allows you to pass a list of string that would act as input constraints, throwing an exception when the user inputs anything outside of those constraints. This allowed for minimal duplication in input logic.

4 Reflection

As an agile team, we all worked together wonderfully, each of us contributing some major aspect of the application design and functionality. Overall we assigned and completed a total of one hundred and four story points, with thirty three of those points being completed by me. Although the story points system was useful to drive conversation, I do feel that it was more of an academic endeavour in this case, as each of us was not overly familiar with the other's technical level, and there may have been some lack of understanding in what a story would require for completion. That being said, over time we became much more confident in each other's and our own ability, moving us toward a more accurate scoring system for the stories as a team. Given more time, we would have been able to figure out our velocity as a team and score stories with much greater accuracy.

As a team, we held daily scrum meetings where we planned what work would be completed that day and assigned ourselves relevant tasks. These scrum meetings also included some level of refinement where we looked over our backlog and modified any stories that would be assigned that day, pushing them closer to the definition of ready. On reflection, it would have been helpful to create a definition of done for many of the stories, as there was some level of confusion where some stories had been marked as done when the backend implementation was ready, without the user interaction layer being set up. This meant that the functionality was not yet ready to be hooked up to the user interface, and an investigation was needed to ensure all backend functionality was eventually represented in the final product.

Sprint planning allowed us to create an initial backlog and map the assignment specification into implementable user stories. We also held a sprint retrospective mid way through the week so that we could reflect on the work completed and see if any blockers were slowing us down and implement help where it was needed. This was massively helpful, with the day of our retro being the most productive of the week, achieving more than half of our total story points completed as a team.

4.1 The Agile Process

Although I am an advocate of the agile process, seeing the benefits and pushing for an iterative approach, I do feel that because of the constrained timeframe, this piece of work would have benefitted from a waterfall-esque approach. For example, there was some confusion around whether certain logic should be held in the service layer or the user interaction layer, but due to time constraints, we felt pushing the conversation further would lead to more time pressure. This resulted in less testable code, and certain methods in the service layer simply calling a method of the same name in the repository layer making the service layer an unnecessary abstraction in certain cases.

Another place I felt the time pressure and agile approach hurting the code quality was in testing. I drove my code with testing throughout Tuesday and Wednesday, achieving high coverage and confidence in my code. When moving away from building totally individual components to working on components that relied on others work, I felt that it was no longer worth the time to rewrite code to be easily tested, so the test-driven approach was abandoned in favour of high velocity. As a result, I could no longer claim confidence in the code, which had to be manually tested and is no doubt fairly brittle to change.

Despite the issues, the team worked well together, and the agile ceremonies were helpful in driving conversation and pushing everyone towards collaboration and honesty. With more time, I believe we would have been able to slow down, iterate in a meaningful way, and find a great working flow to push towards a high quality and reliable piece of software. As it was, we still overcame all of the above issues, completed all requested functionality, and worked together as a team in almost perfect harmony right up until the final moment. I am proud of both my own individual contribution and the overall achievement of the team.