Car Sharing on Demand

Ricardo Duarte

*BSc Hons in Contemporary Software Development* 2024/2025

# 

Computing Department, ATU Donegal, Port Road, Letterkenny, Co. Donegal, Ireland.

Car Sharing on Demand:

A car sharing platform for users and providers

Author: Ricardo Duarte

Supervised by: Pauric Dawson

June 2024

# Declaration

I hereby certify that the material, which l now submit for assessment on the programmes of study leading to the award of Bachelor of Science in Applied Computing, is entirely my own work and has not been taken from the work of others except to the extent that such work has been cited and acknowledged within the text of my own work. No portion of the work contained in this thesis has been submitted in support of an application for another degree or qualification to this or any other institution. I understand that it is my responsibility to ensure that I have adhered to ATU’s rules and regulations.

I hereby certify that the material on which I have relied on for the purpose of my assessment is not deemed as personal data under the GDPR Regulations. Personal data is any data from living people that can be identified. Any personal data used for the purpose of my assessment has been psudonymised and the data set and identifiers are not held by ATU. Alternatively, personal data has been anonymised in line with the Data Protection Commissioners Guidelines on Anonymisation.

I consent that my work will be held for the purposes of education assistance to future students and will be shared on the ATU Computing website (http://www.atucomputingdonegal.com/) and Research THEA website (https://research.thea.ie/). I understand that documents once uploaded onto the website can be viewed throughout the world and not just in Ireland. Consent can be withdrawn for the publishing of material online by emailing Thomas Dowling; Head of Department at jade.lyons@atu.ie to remove items from the ATU Computing website and by email emailing Denise McCaul; Systems Librarian at denise.mccaul@atu.ie to remove items from the Research THEA website. Material will continue to appear in printed formats once published and as websites are public medium, ATU cannot guarantee that the material has not been saved or downloaded.

Signature of Candidate: Date:

# Acknowledgements

I would like to thank…

# Abstract

Written here is no more than 250 words of summary of the problem, hypothesis and main conclusions. The abstract will entice people to read the rest of the document.

# Acronyms

|  |  |  |
| --- | --- | --- |
| Acronym | Definition | Page |
| JWT  V2X  V2P  JVM  BDD  TDD  RSA | JSON Web Token  Vehicle 2 Everything  Vehicle 2 Person  Java Virtual Machine  Behaviour Driven Development  Test Driven Development  Rivest-Shamir-Adleman | 1  1  3  3  3  3  29 |
|  |  |  |
|  |  |  |
|  |  |  |
|  |  |  |

# Table of Contents

[Declaration 4](#_Toc197903620)

[Acknowledgements 5](#_Toc197903621)

[Abstract 6](#_Toc197903622)

[Acronyms 7](#_Toc197903623)

[Table of Contents 8](#_Toc197903624)

[Table of Figures 10](#_Toc197903625)

[Table of Tables 11](#_Toc197903626)

[Table of Code Listings 11](#_Toc197903627)

[1. Introduction 1](#_Toc197903628)

[1.1. Purpose 1](#_Toc197903629)

[1.2. Background 2](#_Toc197903630)

[1.2.1. V2X 2](#_Toc197903631)

[1.2.2. JWT 3](#_Toc197903632)

[1.2.3. Kotlin Multiplatform 3](#_Toc197903633)

[1.2.4. TDD vs. BDD 3](#_Toc197903634)

[1.3. Aims & Objectives 4](#_Toc197903635)

[1.4. Report Outline 5](#_Toc197903636)

[2. Design 5](#_Toc197903637)

[2.1. Requirements 5](#_Toc197903638)

[2.1.1. User Application 5](#_Toc197903639)

[2.1.2. Server 6](#_Toc197903640)

[2.1.3. Car Application 7](#_Toc197903641)

[2.1.4. Database 8](#_Toc197903642)

[2.2. High Level Design 10](#_Toc197903643)

[2.2.1. Use Case Diagram 11](#_Toc197903644)

[2.2.2. Sequence diagrams 11](#_Toc197903645)

[2.3. Low Level Design 15](#_Toc197903646)

[2.3.1. Class diagrams 15](#_Toc197903647)

[2.4. Agile Methodology 19](#_Toc197903648)

[2.4.1. Kanban Board 20](#_Toc197903649)

[2.4.2. Epics 21](#_Toc197903650)

[2.4.3. Stories 21](#_Toc197903651)

[3. Implementation 22](#_Toc197903652)

[3.1. Development 23](#_Toc197903653)

[3.1.1. Compose Functions 25](#_Toc197903654)

[3.1.2. View Model 26](#_Toc197903655)

[3.1.3. Navigation 27](#_Toc197903656)

[3.1.4. RSA Keys and JWT 29](#_Toc197903657)

[3.1.5. Http Client 30](#_Toc197903658)

[3.1.6. Http Server 32](#_Toc197903659)

[3.1.7. WebSocket Server 34](#_Toc197903660)

[3.1.8. WebSocket Client 35](#_Toc197903661)

[3.1.9. Data Classes and Helper Functions 37](#_Toc197903662)

[3.2. TDD 38](#_Toc197903663)

[3.3. CI/CD 40](#_Toc197903664)

[Appendices 43](#_Toc197903665)

[Appendix A: References i](#_Toc197903666)

[Appendix B: Code Listing iii](#_Toc197903667)

# Table of Figures

[Figure 1 Entity Relationship Model diagram 9](#_Toc197902209)

[Figure 2 High Level Diagram 10](#_Toc197902210)

[Figure 3 System use case diagram 11](#_Toc197902211)

[Figure 4 Car Server sequence diagram 12](#_Toc197902212)

[Figure 5 User server sequence diagram 13](#_Toc197902213)

[Figure 6 User Car sequence diagram 14](#_Toc197902214)

[Figure 7 Client App Class Diagram 15](#_Toc197902215)

[Figure 8 Car Application ViewModel and HttpClient class diagrams 16](#_Toc197902216)

[Figure 9 Server Class diagram 17](#_Toc197902217)

[Figure 10 Car service class diagram 17](#_Toc197902218)

[Figure 11 Class diagram for shared data 18](#_Toc197902219)

[Figure 12 Class diagram for WebSocket client and message 18](#_Toc197902220)

[Figure 13 Project’s kanban backlog 20](#_Toc197902221)

[Figure 14 Car Server Jira epic 21](#_Toc197902222)

[Figure 15 Jira story for the creation of the server skeleton code 21](#_Toc197902223)

[Figure 16 Repository structure during initial commits 22](#_Toc197902224)

[Figure 17 Main branch enforced rule set 24](#_Toc197902225)

[Figure 18 Example of a merged pull request 24](#_Toc197902226)

[Figure 19 GitHub Actions successful pipeline from a merge request 42](#_Toc197902227)

# Table of Tables

**No table of figures entries found.**

# Table of Code Listings

**No table of figures entries found.**

# Introduction

Car sharing today presents today, according to (Turoń, 2023) “one of many solutions to help solve mobility in dense urban areas”, sine it aims to provide an easy and affordable solution for short-term trips and allow an individual to easily access a car sharing service over a smartphone in real time. The author of the article also highlights that in 2023 car sharing solutions existed in “over 59 countries offered by hundreds of different providers”, which could highlight the scalability of such solutions in today’s world and evidence how much this numbers could grow soon (JetBrains, 2024) given the trend.

In 2024 in Germany car sharing services “covered more than 1200 cities with over 300 providers”, according to (Wandering, 2024), which highlights the extension of the car sharing services on of the largest EU countries. He also highlighted that car sharing services can be “a valid option for people who don’t own a car or drive sporadically” and that base free “free floating” providers are a great alternative to “more typical car rentals when it comes to short trips” (Wandering, 2024).

Giving that, the purpose of this project is to introduce a generic car-sharing platform, that allows to provide a general, safe and easy to use solution into the car sharing model landscape.

## Purpose

The purpose of this project is to provide a generic unified solution for car sharing with a centralized backend that aims to establish secure sessions between user and car relying on a non-standard user session authentication method instead providing a secure token (JWT) for each session. This solution fits within a V2X design where the vehicle will be able to communicate with both the server to self-register and provide a status update at a constant rate at the same time as it can communicate with the user once a session as been established to execute diverse set of operations part of the car sharing flow (such as car lock and unlock for example).

## Background

The world today is ever more interconnected, especially in urban canters, which provides the need for more ways of transportation and shared mobility are not excluded, for example shared scooters and bikes and cars are among the most convenient forms of transportation, according to (Turoń, 2023), however in the same article the author explains that in the adopted markets the implementation of car sharing services “one can observe the number of closures, takeovers of companies, or implementation of services on only a pilot basis” (Turoń, 2023, p.1185), which indicates that either the available services are monopolized or inexistent on a commercial way, and according to the author, the arguments behind it are: “the system had an ill-matched business model, funds were not properly managed, the project turned out to be economically un-proﬁtable, information found about the ﬂeet of vehicles was improperly matched to the needs of users, customer expectations were not met, or that the scope of operation of the services was incorrectly deﬁned” (Turoń, 2023, p.1186).

Therefore, the purpose of this solution is the implementation of a cross-platform solution that aims to implement a scalable and generic car sharing platform, which allows for adjustment based on the user needs and that provides a seamless and secure of connecting the user to the car, the solution aims to address both the needs of the user and provider by allowing more flexibility in the offer. This project is going to be implemented in Kotlin, with a design based on a V2X which favours the usage of JWT for password less authentication between user and car.

### V2X

The constant evolution of information networks, such as 5G, pushes for to design vehicle wireless technologies capable of communicating with everything from the cloud, people or even other vehicles, and, according to (Hasan *et al*., 2020), V2X as a technology aims to “improve trafﬁc and resource efﬁciency, incidents and road pollution”. However, this technology is not free from challenges and the author mentions that the biggest concerns are regarding “security and privacy”. In this project the idea is to use V2X to connect vehicles to users both through an off-board entity (cloud server) and directly V2P using JWT as a technology for a safe and password less authentication between user and car.

### JWT

According to (Jones, *et al.*, 2015), the standard defines a JSON Web Token as a “URL-safe means of representing claims to be transferred between two parties”, a JWT is composed by an header containing the signing algorithm and type, a payload which represents the data object (in JSON format) being transmitted between entities, and the signature that validates the token itself generated via HMAC-SHA256 algorithm using a private key. According to (Rana, *et al.*, 2023) which also states that “JWT with the HMAC-SHA256 algorithm provides a high level of security in data exchange”, which in this project is used to mitigate the security challenges from the V2X nature of the project. The JWT can also be used for authorization and authentication which will allow for a user to initialize a safe and unique session desired vehicle.

### Kotlin Multiplatform

Kotlin multiplatform is a Kotlin capability that aims for “reducing time spent writing and maintaining the same code for different platforms”, according to (JetBrains, 2024), this allows for the same code to be shared for both web, mobile and desktop while attempting to preserve most of the benefits of native code. Kotlin provides multiple frameworks that will assist this project such as Compose, which is a cross-platform framework that allows developers to easily develop cross platform applications with shared code for both the models (in Kotlin) and UI (declarative). Kotlin also provides KTOR which is a framework for the web, which allows to both write client and server code, KTOR allows to compile for the JVM and natively which provides a great tear of flexibility for the solution being developed.

### TDD vs. BDD

In software development, testing plays a big role as mechanism to ensure that the code behaves or reacts in accordance with what is required and helps to discover and highlight bugs early on in production. TDD “is a test-first software development methodology,” (Moe, 2019), that suggests the creation of a test case before the code implementation, it focuses mostly on unit testing (white box testing) and according to the author “TDD is a low-level, technical methodology that developers use to produce clean code that works.”. On the other hand, BDD “is a customer-focused process. It is based on the full and clear understanding of the system or module behaviour”, therefore tests represent system behaviour therefore the entire team can write tests. For this project given the nature of the team involved (single development) and the amount of separate feature TDD was the chosen methodology.

## Aims & Objectives

The main goal of this project is to develop a car sharing system that focus on simplicity, safety and scalability as well as generalization which aims to provider users and providers flexibility in offer. The main aims for this project are:

* To create a user application that can request a car service to a service provider by:
  + By using Kotlin Multiplatform to create a cross-platform application.
  + By allowing the user to select available cars on demand.
  + By requesting a safe JWT token to a server to create a session with a car and request commands.
* To create a server
  + By using Kotlin KTOR to implement the server-side code and JWT authentication flow.
  + By requesting the car for updated status.
* To create a test car application
  + By using Kotlin native and KTOR for the https client code.
  + By self-registering to a server as a provider.
  + By establishing a connection with a user once a valid request with a valid token is done.
  + By providing the server with up-to-date status.

## Report Outline

Provide a summary of what each of the following chapters will entail

# Design

In order to produce the desired solution introduced in the previous chapter, first a design must be created in order to understand how the project can be implemented, the design section is composed gathering of requirements (both functional and non-functional), the creation of high-level design (flow diagrams), low level design (class diagrams and sequence diagrams) and the necessary agile project management for the implementation phase by creating an agile board with agile artifacts (such as epics and user stories).

## Requirements

When analysing the aims and goals of project is possible to obtain a high-level view of how the system should be but it lacks details necessary for its implementation, therefore this next topic is focused on gathering and categorizing the functional and non-functional requirements which will be the fundamental guidelines for the overall design and implementation.

### User Application

One of the parts of the desired system is a cross-platform application that can ran in multiple platforms, where a user can fetch for available cars, get the car details such as availability, amount of fuel and location for example, and request a session for a specific car, lock and unlock the car and end the current session.

#### Functional Requirements

* [FR000] Be cross-platform application
* [FR001] The application should be able to fetch for cars, from the server and list them to the user.
* [FR002] The listed cars must provide the user the current car status (such as availability, charge or fuel, and exact location) in a list (or map) and cost.
* [FR003] The application should allow the user only to select a car and it must be available.
* [FR004] The application must be able to request the server for a secure token to start a session with a selected car.
* [FR005] The application must allow the user to unlock, lock the car and end a session once a session has been established.

#### Non-Functional Requirements

* [NFR000] The application must be developed using Kotlin Multiplatform (with compose)
* [NFR001] The application must use KTOR framework for the https client-side code.
* [NFR002] The application must use JSON as the data encoding to communicate with both the server and the car
* [NFR003] The application must be able to encode and decode JWT

### Server

The second and most central component of the system is a server application that can fetch data from the car app for the car details, and feed the user application, accordingly, allow to get request from a user to authenticate with a specific car, generate a signed JWT and provide it to the user so it can authenticate with the car.

#### Functional Requirements

* [FR006] Have a server capable of accepting car and user registration.
* [FR007] The server must be capable of providing a public key to a car on registration to validate incoming requests from users.
* [FR008] The server must be capable of providing the user with a secure token on session request.
* [FR009] The server must be able to gather data from registered cars data on a regular basis.
* [FR010] The server must be capable of providing the fetched car data at real time to the user applications registered.
* [FR011] The Server must be capable of storing both user, car, and session data into a database.

#### Non-Functional Requirements

* [NFR004] The server must be implemented in Kotlin.
* [NFR005] The server must use KTOR for the https implementation and token capability.
* [NFR006] The server must use JWT as the technology for secure token.
* [NFR007] The server must respond to car data updates every 30 seconds.
* [NFR008] The server must provide a report back to the client once the cae session has been terminated.

### Car Application

Another part of the system is a car application that constantly sends status data back to the server, it can interact with the user app to initialize a session, receive commands for lock and unlock and end the session on user demand.

#### Functional Requirements

* [FR012] Have an application that can communicate with the server.
* [FR013] The application should be capable to register by itself as a provider to the server.
* [FR014] The application should be capable of receiving a public key from the server to validate future requests from a user.
* [FR015] The application should be able to provide data about its status to the server.
* [FR016] The application should be capable of establishing a session with a user application.
* [FR017] The application should be capable of receiving commands from the user once a session has been established (lock and unlock).

#### Non-Functional Requirements

* [NFR005] The application must be developed using Kotlin native.
* [NFR006] The application must use KTOR as the framework for the https client code.
* [NFR007] The application must be capable of decoding JWT from the client and validate its signature using the public key.
* [NFR008] The application must send a status to the server every 30 seconds.

### Database

In an integrated system storage becomes a reality, a database is required to persist data, otherwise the applications would be dependent on memory, which is tied to the application’s lifecycle, therefore some kind of database is required.

#### Entities

When analysing the requirements, it is evident that both user data and car data represent two different entities in the database, however there is other entities that can be extracted from the requirements such as a car sharing entity and user session entity.

#### Attributes

Entities are composed by attributes, after extracting the entities from the requirements, the next step is the extraction of the attributes of each entity, that could be summarized as follows:

* User should contain a user unique id (used as primary key), a username and an email address.
* Car should contain a vehicle unique id (used as primary key), and car specific data such as fuel type and availability, and price per hour.
* A user session should contain a user unique id (as a foreign key), a valid session flag.
* A car session should contain user unique id (as a foreign key), a vehicle unique id (used as foreign key), the current session JWT and session metadata (composed by time elapsed, current price).

#### Relations

Since all the entities and attributes have been gathered it’s important to understand how they relate to each other, the attributes section already indicates some of this relationship, for example the car session is composed of two foreign keys, that are primary keys of two other entities, which indicates both user and car relate with car session. It’s important to understand both the relationship but also the cardinality (ownership) between them, which can be summarized as follows:

* A User as relation of one to one with user session.
* A User as a relationship of one to one with car session.
* A Car as a relationship of one to one with car session.

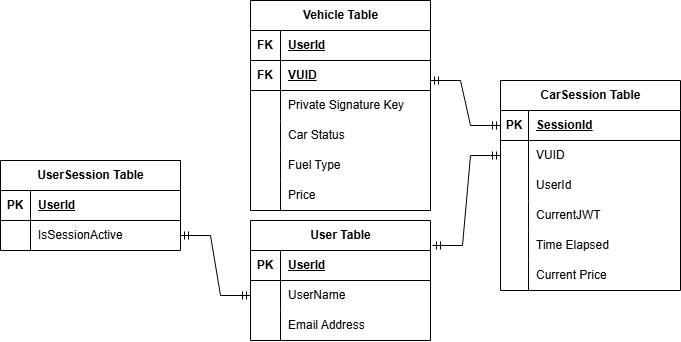


Figure 1 Entity Relationship Model diagram

The image above is an ERM (Entity Relationship Model) diagram the helps to visualize, the entities, the relationships between them and attributes of each entity, which serves as a visual representation of the data model of the database. With this diagram is now possible to create a database for the project, however one important step is still missing which the selection of a database technology. Given the nature of the data being stored and the integration with the KTOR framework, the technology chosen was PostgreSQL, which according to (The PostgreSQL Global Development Group, n.d.), “is a powerful, open source object-relational database” with a “strong reputation for reliability, feature robustness, and performance”, therefore presenting a perfect solution that are aligned with the goals of this project.

## High Level Design

The high-level design dictates a broader image of how the system looks like, which allows for the existence of an overview of what is desired to be implemented which match the defined functional requirements, this was designed using a use case diagram and sequence diagrams.(Moe and University of Computer Studies, Hpa-An, Kayin State, Myanmar, 2019)

A screenshot of a computer

AI-generated content may be incorrect.

Figure 2 High Level Diagram

### Use Case Diagram

The use case diagram provides a broad description of the interactions between the different actors and the designed system. In this case the project main components are represented by three different actors who interact with each other, and each interaction is a use case, this is better highlighted in the following image:

A diagram of a company

AI-generated content may be incorrect.

Figure 3 System use case diagram

### Sequence diagrams

The use case diagram itself only represents the interaction between the actors of the system, which alone do not provide enough information of how the system behaves. In this case a sequence diagram is required to provide information how the system interactions happen on a time sequence, which can provide a sense of linearity and better understanding of the system requirements and how the components of the system interconnect with each other in more detail. In this project three main sequence diagrams where extracted, which can be observed in better detail in the figures that follow:

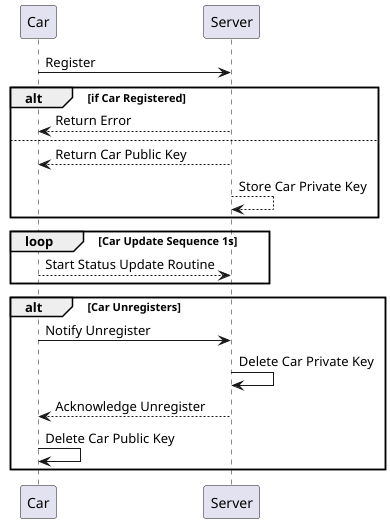


Figure 4 Car Server sequence diagram

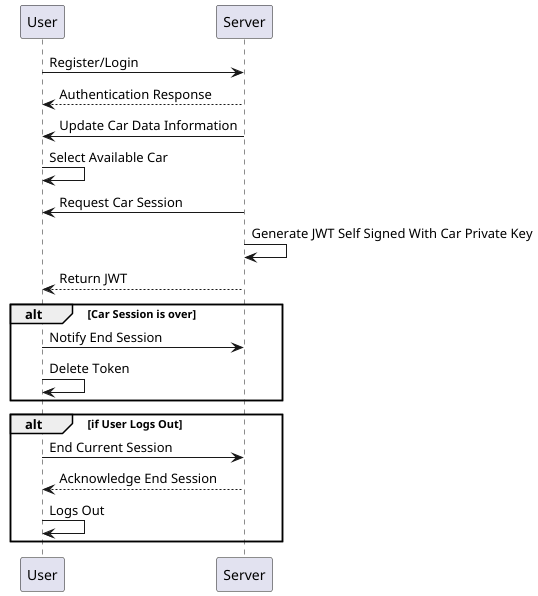


Figure 5 User server sequence diagram

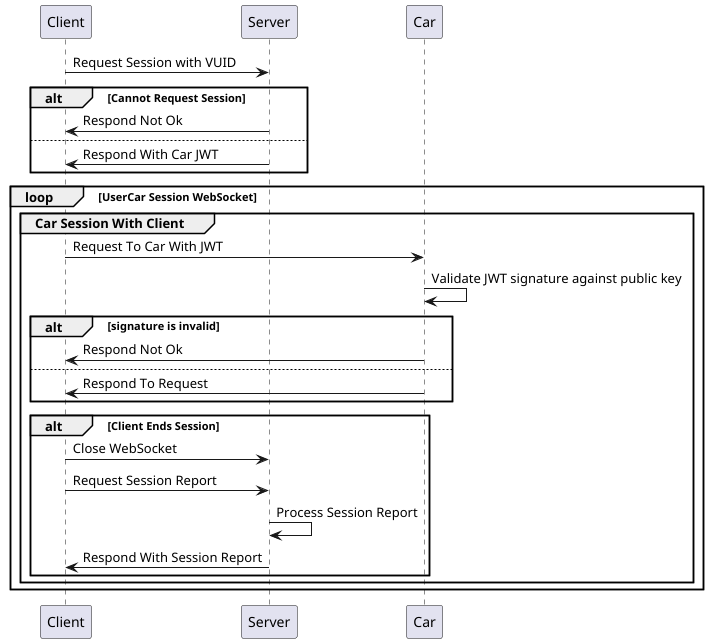


Figure 6 User Car sequence diagram

## Low Level Design

The low level design is a fundamental component of the design chapter which aims to give a more in depth understanding of how the implementation should be, and focus on higher technical details of the development, an example is the creating of class diagrams which is a diagram that represents the components in the system in an object orient manner (units or components represented in classes and instantiated as objects) and how they the relation between them (inheritance, composition).

### Class diagrams

#### Client Application

A diagram of a application

AI-generated content may be incorrect.The first, class diagram is the client application one, which is mostly defined by composable functions given the dependencies to the Kotlin compose framework, given that the class diagram is mostly composed by functions that define the different views of the application:

Figure 7 Client App Class Diagram

The diagram above mostly evidences the design of the UI components, however, best practices define that GUI applications should require the separation of UI (View) from logic (Models), therefore the is the need for a separate class to handle the state and the connection between data and UI (ViewModel).

The last big component defined in the requirements for the client application. Is the handling of communication with the server via HTTP (HTTPs ideally), which requires the creation of an HTTP client code that process to makes HTTP requests to the server and handles the responses. Given that the HTTP client is a different unit (single responsibility principle) it justifies the design of a class that wraps all the HTTP logic.

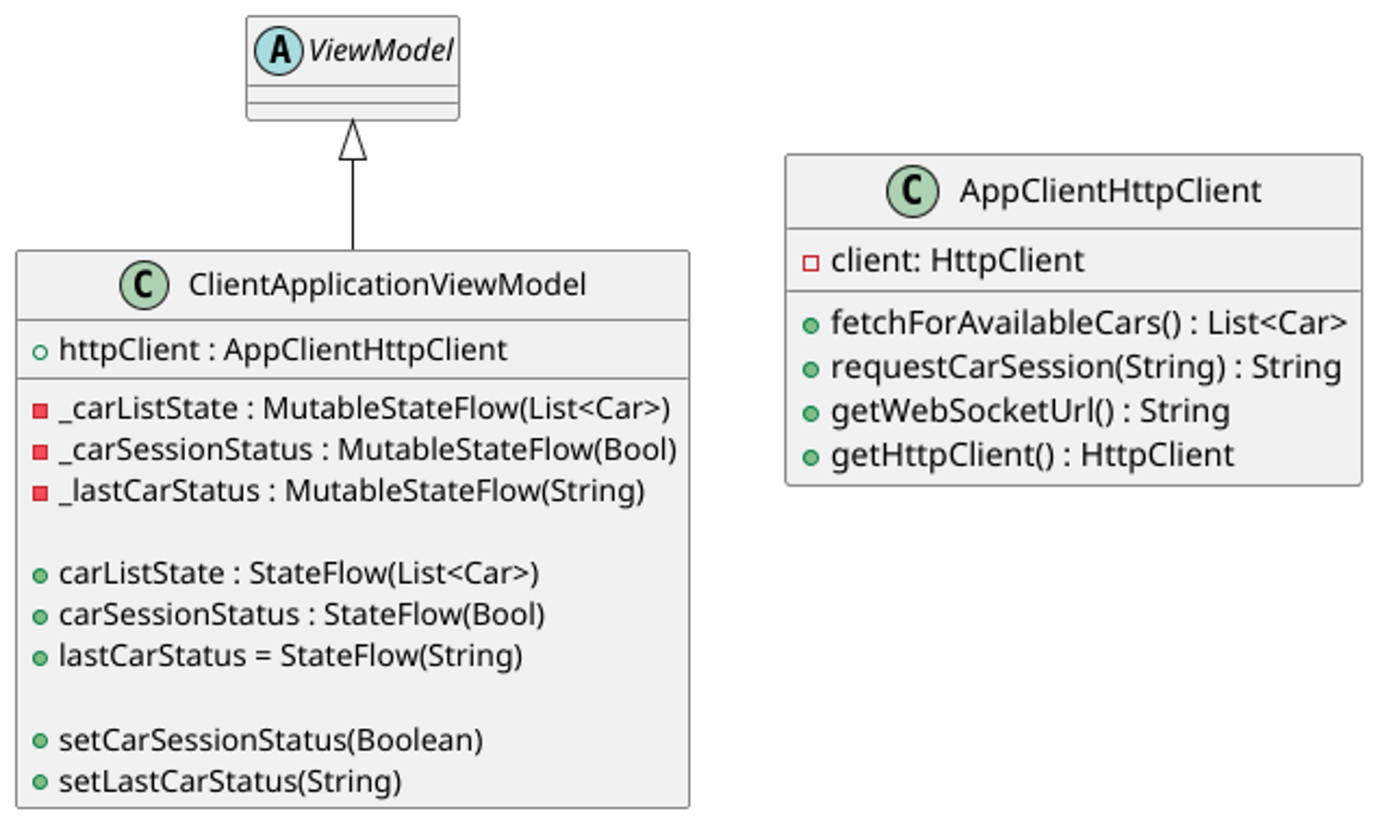


Figure 8 Car Application ViewModel and HttpClient class diagrams

#### Server

Unlike the client application the server does not require any UI therefore all logic should be centralized in one place which is means the class diagrams mostly evidences the definition of the routes a class to hold the server state and the main class that is composed of the routing object and a starting method as visible in the image below.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 9 Server Class diagram

#### Car service

The car service was designed as a standalone binary that like the client application requires a client to handle HTTP requests but since it only handles request and responses from the server all the logic can be contained in one class that is composed by the HTTP client object, a status object and some methods to handle the requests and response using the HTTP client object.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 10 Car service class diagram

#### Shared

Since both clients share the same data for data interchange (like RSA keys), both requires creating a similar WebSocket logic some of code needs to be shared. To avoid code repetition, it was important to design some shared code and logic that can be shared with other parts of the project which can be seen in the diagrams bellow:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 11 Class diagram for shared data

A screenshot of a computer

AI-generated content may be incorrect.

Figure 12 Class diagram for WebSocket client and message

## Agile Methodology

Given the nature of the project being developed and its complexity, some form of project management is required so the development can meet the needs of requirements as much as possible. As far as methodology is concerned there are major methodologies in software development today, such as waterfall which according to (Falade rhoda, *et al.*, 2025) focus on “Detailed upfront planning, fixed scope, phased progression” but “Waterfall resists change, risking irrelevance” and “Scope issues surface during testing, inflating costs if misaligned with needs”, on the other end agile which key feature are “Iterative planning, continuous stakeholder feedback, adaptive scope refinement” while it risks “Over-Flexibility: Without discipline, Agile can invite endless changes”. Given the pros and cons of both methodologies the one selected for this project was the agile methodology since it provides higher flexibility deliverable increments and nature of the project helps mitigate ant risk of endless change given its straightforward nature. Within the agile methodology there exist several frameworks such as Kanban, SCRUM, SAFe, etc.., for this specific project Kanban was selected given the size of the team and nature of the segments of work (linear workload).

### Kanban Board

Kanban as many other frameworks, presents its artefacts in a board called Kanban board, in which columns separate the flow of work, in this project the flows defined were, *Ready* which represents any story or task that contains a complete description and is ready to be started, *In Progress* which holds any story or task in active development and finally *Done* which represents every task which have been merged to the main branch. The project’s board can be visualized in the figure below:

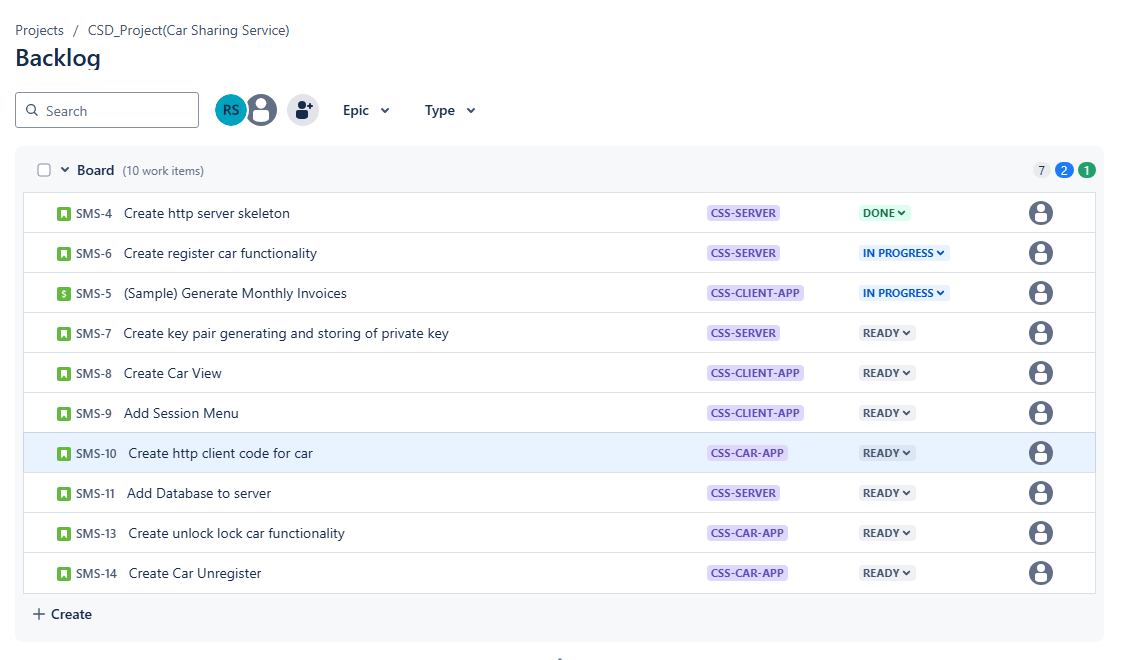


Figure 13 Project’s kanban backlog

The project’s JIRA board [[1]](#footnote-1) is used to not only as a project management tool, but also to track down code changes to issues (tasks, or stories) to provide a level of traceability between implementation and the requirements. Additionally, it provides a quick and easy way to pinpoint change in time if case a bug occurs or even to link test cases to use cases (for quality assurance).

### Epics

An epic is an agile artifact that represents a large feature, usually an epic is written in a very high-level perspective mostly focused on a larger scope of the feature, highlighting the feature’s functional requirements. This project contains three epics Server, User App and Car App. The image below is of one of the project features:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 14 Car Server Jira epic

### Stories

Stories in agile represent a smaller part of a feature, usually carries a more granulated description written in the perspective of the developer or user and aims to implement one or more functional requirements. The figure bellow represents one of many tasks of this project:

A screenshot of a computer

AI-generated content may be incorrect.

Figure 15 Jira story for the creation of the server skeleton code

# Implementation

The implementation phase is one of the most important phases of the project, which is the stage in which the code is written, and the first palpable results of the project start to appear, it defines the step in which the project crosses the theoretical component to the practical component. When analysing the design, it’s possible to understand that the solution being implemented is composed of three major components (three different executables) the development couldn’t be done separately which required that a strategy to be defined before starting to write code. The early stages of the development were marked by the organization and planning of the work to be done, which was done by refining the existing tasks in the Kanban board and the addition of new stories when need as a product of the refinement. The focus was to distribute the development into features that could be developed and tested in isolation. The second step was the creation of a repository on GitHub for source and version control. The initial commits of the repository were initially composed of what is considered skeleton structure of the project, which was composed of the barebone files of the project.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 16 Repository structure during initial commits

As visible in the image above the initial commits were focused on creating a project structure, which defined that the project was divided in four sub-projects *car\_demo\_app* (not yet visible in this image), *composeApp*, *server* and *shared* which match the logic separation defined in the low-level design. The figure also evidences that the project is done 100% in Kotlin, which matches the non-functional requirements defined in the prior stage, given that fact and the structure of the project, the build system used for this project was Gradle which is an “open source build system of choice for Java, Android, and Kotlin developers. From mobile apps to microservices…” according to ( Gradle, Inc, 2019), gradle allows for the creation of scripts for jobs (such as linting, building, testing, etc.), in a Kotlin same language as the code itself an allows for integration with a CI/CD configuration, justifying this way the choice of Gradle over Maven for example.

## Development

In this phase the project already contains a repository and a build system in place, which are the necessary baselines to start developing the code. As mentioned before the Kanban board was refined at this point and was mostly composed of stories that represented individual features that could be done in isolation. Since the project is composed of three major components and shared logic and data the development of this components could never be done in a sequential manner but had to be done on a requirement per requirement manner. During this stage a working model was defined so that it was ensured that the code being developed was always buildable (no compilation failures) and testable (all logic unit tested) which aimed to respect continues delivery as much as possible. For that it was defined that the main branch of the repository would be protected and defined as the default branch, this required to create a ruleset on GitHub that forbids direct addition of code to main (direct pushes), forces the creation of a feature branch and the creation of a pull request to merge it once all jobs succeed (later discussed in the CI/CD section) and development is complete, ensuring mainline is always in a stable state.

A white line with blue lines

AI-generated content may be incorrect.

Figure 17 Main branch enforced rule set

The working model above mention was directly influenced by Trunk Based Development, that according to (Paul, n.d.), it suggests code can be “directly to trunk commit/push or a Pull-Request workflow as long as those feature branches are short-lived and done solo or pair programmed“, self-proclaiming to be a “key factor for Continuous Delivery and Continuous Integration” which ensures the that “ the codebase is always releasable”. Therefore, most of the development of the code respected this defined working model by adding.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 18 Example of a merged pull request

Other than the working model, another important aspect of the implementation is the actual code written in the project. This section does not aim to cover every single line of code written, but to highlight and explain the primary components of the project, those contain the bulk of the logic developed and provide a better understanding of implementation as whole and how all features interconnect as well as how they were translated to source code.

### Compose Functions

During the design process, it was defined that the client application was to be written in Kotlin using a framework called Compose Multiplatform[[2]](#footnote-2) that allows to develop cross platform applications using a declarative UI structure to write the application components (very similar to widgets), being developed by JetBrains the company that create Kotlin and being currently at version 1.8.0 becoming the facto choice for the client application. Compose multiplatform does treat every component as a composable function, this means a screen is a function as well as a button for example.

A computer screen shot of a program code

AI-generated content may be incorrect.

Code 1 CarListScreen Compose function code snippet

The image above shows a compose function, this snippet of code implements a UI component that lists of available cars (in a Card Format). The functions are annotated with Preview (optional) and Composable (required) which are two specific annotations for the framework that define that the function is a composable function and can be previewed by the preview tool in the IDE. The body function calls on a Scaffold function (another UI component that can hold other components and child components), inside Scaffold is a Column function that defines a component that as children aligned in a vertical stack, these functions accept specific parameters such as Modifiers, that define the visual of it (such as alignment for example). The inner components of the list are a Text component and CarCards (custom component based on default Card) of the available cars fetched from the server. The remaining views of the project were done in the exact same way by nesting compose functions and modify them to achieve the desired component.

### View Model

The ViewModel is the class that connects the View (UI) with a model (Data), in the client application only one ViewModel was created, and it’s shared between the different views.

A screenshot of a computer code

AI-generated content may be incorrect.

Code 2 ViewModel code snippet

The code snippet above highlights the class the connects the car data being fetched from server and the UI, the case of Compose Multiplatform[[3]](#footnote-3), by default the views are static this means that the data displayed on the screen does not suffer mutation by default, for the application to react to data change MutableStateFlows are required, which work as observables to a value and then the *remember* API is used to react to the values being observed, by storing it in the composition and returning it in subsequent recompositions. The ViewModel also stores the http client used by the client application and it spawns a coroutine[[4]](#footnote-4) (a lightweight alternative to threads that allows for the suspension and resume of a routine) on construction (init block) that fetches for car data every 500ms.

### Navigation

Kotlin Compose does not offer a very intuitive way to navigate between different views, so navigation was a real challenge using the selected framework for the client application, in order to have navigation working the primary screen must create a NavHost which is a composable function that aggregates and centralizes navigation logic, and it requires the registration of the navigable views in it by providing a static serializable data type as a templated parameter and the instantiation of the view as well as any desired parameters for it.

A screenshot of a computer code

AI-generated content may be incorrect.

Code 3 Navigation registration code snippet

A screenshot of a computer code

AI-generated content may be incorrect.

Code 4 Navigation serializable types

The images above evidence the explanation provided before *Code 3* shows the NavHost function and registration of the views and Code 4 show the serializable types and parameters used to identify the views during navigation. To actually navigate between views the origin views either require to receive the navigation controller object (that interacts with the NavHost) or a the navigation controller navigate method is passed as a callback to later be invoked as an action on user interface.

### RSA Keys and JWT

One of the requirements and primary goals of the project was to use JWT to authenticate client requests to car, Kotlin KTOR[[5]](#footnote-5) framework provides code to generate, sign and verify JWT. A JWT[[6]](#footnote-6) is composed of three parts, a header that contains information such as the algorithm used for the JWT signature and the type (usually “JWT”), it contains a payload that is a user defined JSON object being sent to client and the last part of the JWT is the actual signature of the token, that ensures that the token was emitted by a trusted entity. In this implementation the token was signed using a symmetric algorithm (RSA) which provides two signature keys, a private key that signs the JWT and a public key (shared with the entity that validates the JWT) used to validate the signature.

A close-up of a white background

AI-generated content may be incorrect.

Code 5 JWT creation and signing code snippet

A screenshot of a computer code

AI-generated content may be incorrect.

Code 6 RSA key pair generation code snippet

The code snippets above highlight the creation of the JWT as explained and the usage of the standard security library (from Java but 100% interoperable with Kotlin) to generate the RSA key pair used to sign and validate the JWT.

### Http Client

Both the car service and the client application in the project interact with an HTTP server, by making HTTP requests and receiving its response. KTOR[[7]](#footnote-7) was the framework used that provides which allows to create an asynchronous HTTP client to interact with the server, the client is created by instantiating an HttpClient object and providing an engine (by default KTOR uses the CIO engine but others are available). In this case, the class was designed to allow for dependency injection, so that test engines (mocks) can be provided when testing the HTTP logic. The HTTP client allows during instantiation to install plugins, plugins are specific optional features that can be enabled, in this case the installed plugins were logging (which provided a default logger for the client) and WebSockets used to create WebSocket sessions to receive requests and send responses to the client.

A screen shot of a computer code

AI-generated content may be incorrect.

Code 7 Http client instantiation code snippet

The image above shows how the client is instantiated, the object is a composition part of a wrapper class that wraps the requests in methods to provide a level of abstraction, to allow scalability (new requests, added to new as methods) and testability.

The HTTP requests are wrapped in methods that can be called by the viewmodel. This methods are required to be suspended and therefore executed in a coroutine scope because KTOR does implement the HTTP client and server using coroutines, which means that HTTP requests and responses are done asynchornosly in their own coroutine space, the inner http client object provides methods that are 1:1 matches with the HTTP methods (GET, POST, PUT, DELETE, etc.) which require a url (mandatory) which can contain parameters and in this implementation all car requests required a query parameter with the vuid value of the car being targeted in the request, the request could contain a body (optional) which usually carrys date to be handled by the server. The snippet of code below shows the implementation of the get request to request a session that creates a get request by providing the url with the vuid query parameter, evualates the response (by checking the HTTP status code returned in the response) and returns the response body as text, in this case it contains the JWT to be used on the requests to the car.

A screen shot of a computer code

AI-generated content may be incorrect.

Code 8 Request car session method code snippet

The car service HTTP client is implemented in the exact same way and follows the same approach to wrap the requests.

### Http Server

Similarly to the client code KTOR[[8]](#footnote-8) also provides in its framework the necessary tools to create the server-side implementation, but unlike the client-side, for the server KTOR provides an Application object (Kotlin specific single instance class) that allows the developer to define a module in which plugins are installed, and the routes are defined. This module is later instantiated in the server main invocation alongside of the server URL and port. KTOR allows to create either an embedded server (defined in code) or Engine Main that uses YAML to configure it.

A screenshot of a computer program

AI-generated content may be incorrect.

Code 9 Server instantiation code snippet

The routing of the server can be done directly inside of the routing block or extracted and invoked in the same block. For this implementation the route definitions were extracted to allow for single responsibility principle to be applied by treating the routes as individual functions with individual responsibilities, to allow for better testability and scalability as well as uncluttering of the module function.

A screenshot of a computer code

AI-generated content may be incorrect.

Code 10 Server route for car status update code snippet

In the snippet above the route is a method of the Route function that handles a POST request from the car to update it´s status, which requires a vuid as query parameter and checks it against the map of registered cars (only registered cars can update their status), updates the state of the car if succeeds on the validations and returns an OK otherwise it returns a bad request HTTP status code.

### WebSocket Server

To provide a full duplex communication, a WebSocket was required, the HTTP server also provides the capability to create WebSocket, and KTOR[[9]](#footnote-9) provides the required tools to implement it. In this implementation, was created so that both clients could interchange communications, the client by making requests to the car and the car by responding back to the requests, in this case the server only mediates this communication by redirecting them. Other strategies such as long polling or server side events could be used but they offer tremendous disadvantages such as long polling practically suspends a request until further data appears so the server can then respond this would required defining delays between requests and responses which would impact reactivity and user experience and server side events do not allow for client to respond back which would require extra routes for dedicated responses to server events. The WebSocket works just like another route in this implementation a WebSocket is open, and the user only connects once a session as been requested and the both the server and clients use a SocketMessage which is a custom serializable data class that is composed of *receiver* and *payload* and it’s serialized to JSON the server then reads the message and redirects to the receiver’s corresponding client.

A screenshot of a computer code

AI-generated content may be incorrect.

Code 11 WebSocket server code snippet

The code snippet above highlights the explanation provided previously, the way the HTTP server separates HTTP requests from the WebSocket is by listening to the WebSocket on a URL with the prefix ws:// instead of http://, and in order to handle multiple session (between different client and different cars) the WebSocket URL must contain the vuid as suffix and an entity query parameter that defines if the client is the car or the user, the server only allows for two entities per session one car and one user.

### WebSocket Client

The WebSocket Client unlike the server, in this implementation it’s wrapper in a class that receives the HTTP client (responsible to create the socket session) and provides methods to create socket which must provide the same URL mentioned in the server and all the parameters, it provides a method to send data and receive data.

A screen shot of a computer code

AI-generated content may be incorrect.

Code 12 WebSocket client code snippet

The WebSocket client is both used by the car service and the client application as mentioned above and they both share the same class, but they instantiate their own instance of it by providing their own instance of the HTTP client to create and handle the socket session. Just like the HTTP request the WebSocket handling is also done via suspended functions that require to be invoked in a coroutine scope.

A screenshot of a computer code

AI-generated content may be incorrect.

Code 13 Usage of the WebSocket in the client app

A screenshot of a computer code

AI-generated content may be incorrect.

Code 14 Usage of the WebSocket in the car service

### Data Classes and Helper Functions

As mentioned in the beginning of this module, there a section of the code that is shared amount the major components (sub projects), the WebSocket client mentioned early is one of the examples of the shared code, which is used by both the car service and client applications, this removes the need for code duplication and reduces complexity. Other examples include the WebSocket message definition, the Car data class, the RSA Key Pair (already mentioned early) and other data classes and helper functions used extensively in the project.

## TDD

During the introduction stage it was mentioned two different methodologies TDD and BDD. For this project TDD was adopted as the methodology during implementation, which mostly focused on writing tests that validate happy path and corner cases and then implementing the code in expectation that the tests would pass after implementation. The decision to use TDD was done so that code being merged to mainline is not only buildable but also tests, which increases the level of confidence in the product but also allows to detect bugs and issues sooner (during development phase). The tests written prior to the code were unit tests, and as the name implies unit tests test units of code, this means features in isolation are tests and it’s a type of white box testing where the implementation is known. In this case, the implementation is only done after the test, but the test provides an idea of the code to implement. The unit tests are written using the Kotlin test plugin[[10]](#footnote-10), and the design of the test is done by creating a fixture class (that tests a specific feature) and then methods annotated with test are added to test different cases. Every case aims to test the flow and behaviour of the unit under test, for example the happy path case tests the main flow of the code as the name implies and then result is asserted against the expected result, the alternative flows are also tested and are also asserted against the designed defensive results given in the implementation.

A screenshot of a computer code

AI-generated content may be incorrect.

Code 15 Test Fixture and happy path case code snippet

A computer screen shot of a program code

AI-generated content may be incorrect.

Code 16 Corner case scenario code snippet

The snippets showcase a unit test of the car status route, and in the first picture the test runs the happy path case in which the car is registered, and it makes a POST request to the correct URL providing the necessary query parameters and receives an OK status code. In the second image, the snippet shows a test scenario where the car is registered and makes a POST request to the correct URL, but it does not provide the necessary quey parameter and expects to receive a bad request response. In this case to test the client is mocked so that the code is truly tested in isolation. More details on testing and testing cases are later provided in the testing chapter this only attempts to highlight the TDD approach taken during the development.

## CI/CD

GitHub provides an integrated tool for CI/CD called GitHub Actions [[11]](#footnote-11) that allows for the creation stages composed by automated jobs that run in accordance with a defined configuration using YAML as the markup language to configure it. Given the working model defined for this project the stages defined (workflows) are triggered on a merge request which initially runs a code scan and linting, and if the jobs succeed then the pipeline invokes the building jobs that build the client app for Android, Desktop (MacOS, Linux and Windows), and the car service and server for Linux. On success the build artifacts are provided to the last job that run the tests.

A screenshot of a computer program

AI-generated content may be incorrect.

Code 17 Snippet of the lint workflow for GitHub Actions

The GitHub actions use the same gradle scripts that are used locally to lint, build and test, and they allow the automation of the different stages of development. The snippet above shows part of the workflow file used for the linting job, and it creates a workflow called lint that is called when the build workflow calls it (which is invoked every merge request and subsequent code pushes), and defines a job (the second is not visible but shares the same structure) named AndroidLintJob, that runs on the latest version of the ubuntu runner (internal container) , which checkout the code, setups java and gradle and runs gradle lint command to invoke the gradle android linter plugin, and then outputs the report artifact with the name android-lint-report extracted from the lint result path. All the other workflows follow a similar structure to this one.

A screenshot of a computer

AI-generated content may be incorrect.

Figure 19 GitHub Actions successful pipeline from a merge request

# Appendices

# Appendix A: References

Falade rhoda, A., Philip, T. & Ava, A., 2025. Agile vs. Waterfall: Which Methodology Best Reduces Scope Creep?. p. 5.

JetBrains, 2024. *Kotlin Help.* [Online]   
Available at: https://kotlinlang.org/docs/multiplatform-intro.html  
[Accessed 2025].

Jones, M. B., Bradley, J. & Sakimura, N., 2015. *JSON Web Token (JWT).* [Online]   
Available at: https://www.rfc-editor.org/rfc/rfc7519.html  
[Accessed 2025].

Moe, M. M., 2019. Comparative study of test-driven development tdd, behavior-driven development bdd and acceptance test–driven development atdd. *International Journal of Trend in Scientific Research and Development,,* 3(4), pp. 231-234.

Monowar, H., Sibin, M., Takayuki, S. & Hongsheng, L., 2020. Securing Vehicle-to-Everything (V2X) Communication Platforms. *IEEE Transactions on Intelligent Vehicles,* 5(4), p. 22.

Rana, M., Pandey, A., Mishra, A. & Kandu, V., 2023. Enhancing Data Security: A Comprehensive Study on the Efficacy of. *International Journal on Recent and Innovation Trends in Computinng and Communication,* 11(9), pp. 4409-4416.

The PostgreSQL Global Development Group, n.d. *PostgreSQL: The World's Most Advanced Open Source Relational Database.* [Online]   
Available at: https://www.postgresql.org/  
[Accessed 2025].

Turoń, K., 2023. Factors Affecting Car-Sharing Services. *Smart Cities,* p. 17.

Wandering, S., 2024. *Carsharing 2024: Providers Compared.* [Online]   
Available at: https://www.navit.com/resources/carsharing-2024-providers-compared  
[Accessed 2025].

# Appendix B: Code Listing

[Code 1 CarListScreen Compose function code snippet 25](#_Toc197902144)

[Code 2 ViewModel code snippet 26](#_Toc197902145)

[Code 3 Navigation registration code snippet 28](#_Toc197902146)

[Code 4 Navigation serializable types 28](#_Toc197902147)

[Code 5 JWT creation and signing code snippet 29](#_Toc197902148)

[Code 6 RSA key pair generation code snippet 29](#_Toc197902149)

[Code 7 Http client instantiation code snippet 30](#_Toc197902150)

[Code 8 Request car session method code snippet 31](#_Toc197902151)

[Code 9 Server instantiation code snippet 32](#_Toc197902152)

[Code 10 Server route for car status update code snippet 33](#_Toc197902153)

[Code 11 WebSocket server code snippet 34](#_Toc197902154)

[Code 12 WebSocket client code snippet 35](#_Toc197902155)

[Code 13 Usage of the WebSocket in the client app 36](#_Toc197902156)

[Code 14 Usage of the WebSocket in the car service 37](#_Toc197902157)

[Code 15 Test Fixture and happy path case code snippet 39](#_Toc197902158)

[Code 16 Corner case scenario code snippet 39](#_Toc197902159)

[Code 17 Snippet of the lint workflow for GitHub Actions 41](#_Toc197902160)

1. https://atu-team-dktaegt8.atlassian.net/jira/software/projects/SMS/boards/1 [↑](#footnote-ref-1)
2. https://www.jetbrains.com/help/kotlin-multiplatform-dev/get-started.html [↑](#footnote-ref-2)
3. https://developer.android.com/develop/ui/compose/state [↑](#footnote-ref-3)
4. https://kotlinlang.org/docs/coroutines-overview.html#how-to-start [↑](#footnote-ref-4)
5. https://ktor.io/docs/server-jwt.html [↑](#footnote-ref-5)
6. https://jwt.io [↑](#footnote-ref-6)
7. https://ktor.io/docs/client-create-new-application.html [↑](#footnote-ref-7)
8. https://ktor.io/docs/server-create-and-configure.html [↑](#footnote-ref-8)
9. https://ktor.io/docs/server-create-websocket-application.html [↑](#footnote-ref-9)
10. https://kotlinlang.org/api/core/kotlin-test/ [↑](#footnote-ref-10)
11. https://docs.github.com/en/actions [↑](#footnote-ref-11)