# Aims and Objectives

The goal of this project is to develop a web-based application which supports a car sharing (short-term car rental) system in which clients (users) can submit requests and the system provides an optimal solution in allocating cars to users.

The aim of the project is to gain insight into the algorithmic difficulties of the problem and to come up with a software system that produces good solutions.

The main objectives of the project:

* Research, understand and describe current works on car sharing algorithms and optimization of said algorithms.
  + Information gathered from relevant sources and from related industries will be described in this study.
* Suggest methods for allocating resources to customer requests.
  + Using the information gathered as a base, I will build up my own ideas on ways ‘quality’ of a solution can be identified and measured fairly between different techniques.
  + An emphasis will be placed on techniques that aim to maximize profit by serving the most customer requests with the available resources.

Challenges presented by the project:

* Understand and model a complex optimization problem arising in a practical application and come up with effective solution approaches.
* Design and implement optimization methods and evaluate them systematically in computational experiments or using theoretical analysis.

Learning outcomes:

* Learn to design and implement software systems that optimize the allocation of resources to customer requests.
* Learn to apply combinatorial optimization methods to problems arising in real-life applications.
* Learn to implement combinatorial optimization methods and evaluate them systematically in computational experiments or using theoretical analysis.
  + Although not covered in detail, an understanding of how this could be used to improve solutions for this project.

In addition, the advanced aims which are desirable, if time is permitting, are:

* To implement an informative and interactive map-based user-interface. This would allow the user to be able to see what is happening live and could enhance the user experience and usability of the software.
  + Learning to use the Node.js JavaScript library which is popularly used to in implementing the Google Maps API.
* To Implement optimized algorithms (evolutionary algorithms) that improve the resource allocation problem over time as more and more requests are made to the system.

# Requirements

## Functional Requirements [F]

**[F0]** User should be able to submit a request for a car:

* **[F1]** User must be able to enter their pickup/starting location **(where)**
  + User must be able to enter their pickup/starting time **(when)**
* **[F2]** User must be able to enter their drop-off/ending location **(where)**
  + User must be able to enter their drop-off/ending time **(when)**
* **[F3]** User must be informed if their request has been sent to the system (loading graphic or animation).
* **[F4]** User must be informed if their request will be accepted/fulfilled (once accepted, there must be a car at available to that user at the requested time) or declined. **(yes/no)**
* **[F5]** User must be given the location and details of the car they will be using once their request has been accepted. **(where)**
* **[F6]** User must be given instructions on how to access the car once the request has been accepted. **(how)**

## Recommended Requirements [R]

* **[R1]** User should be able to create/register an account.
* **[R2]** User should be able to securely login into the system using a username and password.
* **[R3]** User must be able to change their password.
* **[R4]** User must be able to recover or reset their password.
* **[R5]** User should be able to pick from a list of cars (sorted from nearest to farthest) if there are multiple available cars near the pickup location.
* **[R6]** User should not be able to rent a car for more than one week at a time.
* **[R7]** User should be able to see their own activity and history.

## Quality Requirements [Q]

### Security

* **[Q1]** User passwords should never be stored as plain text - employ some form of hashing.
* **[Q2]** User payment information/details should never be stored as plain text.

Performance - response times/responsiveness

* **[Q3]** User should be informed if their request has been accepted/declined within 30 seconds. **(Ideally)**
* **[Q4]** The application must display the front page within 5 seconds of the user logging in.

### Learnability and Usability

* **[Q5]** Most users should be able to request a car on the app the first time without assistance. **(Easy to use)**
* **[Q6]** User should only be notified of an error when it is critical – i.e. error handling, only inform the user when all attempts have been made to prevent this error. **(Intuitiveness & difficult to break)**
* **[Q7]** The system will have a feature for customizable notification settings.

### UI Accessibility

* **[Q8]** Language used should be easy to comprehend. **(Readability)**
* **[Q9]** Avoid certain color combinations to accommodate colorblind users. **(UI colors)**

## Optional Requirements

* User-friendly front-end interface implementing Google Maps API that shows the location of cars:
  + Available cars should be marked with a certain color.
  + Unavailable cars should not appear on the map.
  + Cars that will be available soon (~15 mins should be marked with a certain color).
* Admin features
  + Admin should be able to create, view, edit, and delete user accounts from the system.
  + Admin should be able to add/remove cars from the system.
  + Admin should be able to reset user passwords, sending a password reset email.
  + Admin should be able to view/change the status of cars in the system.

# Description of Prototype

A web application has been implemented, which allows users to request cars for short-term rentals. The web application uses the model-view-controller (MVC) architecture built on Spring MVC. MySQL is used for the relational database.

The application uses some jQuery client-side validation to reduce the server’s validation load and improve the application’s performance. The amount of client-side checks/validation will be increased to make the server’s load as light as possible while simultaneously refactoring server-side validations. A goal is to optimize the application’s speed, responsiveness, and overall performance to improve the user experience.

As per the requirements, passwords are never stored as plain text. Spring Boot provides a built-in BCrypt function called ***BCryptPasswordEncoder***. BCrypt is an adaptive, random hashing algorithm that combines a Salt as part of the hashing process. The combination of a salt and a hash allows us to follow security best practices [12]. BCrypt “*allows us to build a password security platform that scales with computation power and always hashes every password with a salt*” [5].

## Features Implemented

* Registration
  + **[R1]** Users can register using their first & last names, email and password.
* Login & Security
  + **[R2]** Users can login securely using their email and password.
  + **[Q1]** User passwords should never be stored as plain text - employ some form of hashing.
* Password Reset
  + **[R4]** Users can reset their password if they have forgotten.
* User features
  + **[F1]** Users can submit a request for cars by inputting their location, start & end times.
  + **[F5]** User must be given the location and details of the car they will be using once their request has been accepted.
  + Users can cancel currently active requests.
  + Users can view the details of their accounts.
  + **[R3]** Users can change their passwords while logged in.
  + **[R7]** Users can view a summary of their history/past activity.
    - Users can further view all the details of each rental.
* Admin features
  + Admin can view the details of the system.
  + Admin can create/view/update/delete user accounts.
  + Admin can reset user password, sending an email to their email address.
* Email service
  + Sends emails when users register, change/reset their password.
* Error handling/Restricted Access
  + Users are shown a 404 page and redirected to the dashboard if an error occurs.
  + Users do not have access to features they are not authorized for, i.e. user cannot access admin features.

## Features to be Implemented

* Payments & Invoices
  + Generate invoices for fulfilled user requests.
  + Integrate a payment system (preferably PayPal or Stripe API).
* Account Activation
  + Users will be able to activate their accounts by entering their driver’s license, address and phone number.
* User features
  + Users will be able to change their account details such as email, phone number.
  + Users will be able to enter their address.
  + Users will be able to delete their accounts.
* Admin features
  + Admins will be able to add/view/update/delete cars to the system.
* Better exception handling
  + More custom exceptions and handlers.
* Haversine Formula to calculate the distance between user’s desired pickup location and a car’s given location.
* Algorithms for Car-Sharing Problem
  + Develop new algorithms that attempt to solve the car-sharing problem in the application’s given scenario.

## Summary

Going into semester 2, learning more of the jQuery library is a high priority as it will be needed for implementing upcoming features such as an improved Timepicker in the user dashboard as well as improving the speed and performance of the software product (reducing server-side load for processing/requests). Expertise in working with HTML, CSS and Bootstrap also needs to improve as it remains the most time-consuming aspect of developing the software product. Another area of focus for semester 2 is the design of algorithms for the car-sharing problem which I discuss further in the following section.

New bugs and unexpected behavior were observed during the prototype demonstration due to a lack of unit and integration testing. To avoid a reoccurrence and ensure the application is properly tested, I will be employing automated testing as well as manual testing and provide documentation following a template found under ***/docs/5\_manual\_tests***. Automated testing will follow the best practices laid out in the Spring Boot reference documentation [3], which follows the F.I.R.S.T (fast, independent, repeatable, self-validating, timely) principle. Moving forward, I intend to apply a more test-driven development methodology to speed up implementation and testing of features.

I have completed Junit testing on the less complex parts of the system such as domain and utility classes. Junit testing will not be enough for testing more complex parts of the application such as controllers, validators, exception handlers, and repositories. These will be tested using mock objects. I initially planned to test these using the Mockito framework but found that the Spock Specification framework [9] is easier to use and more flexible. Using Spock, I have performed unit and integration testing on all controllers and validators. Features based on functional requirements will not be heavily tested until the application is finalized as these are constantly being reworked and refined.

# Software Architecture, Algorithms and Data Structures

## Software Architecture

The web application uses the Model-View-Controller (MVC) architecture built on Spring MVC and Spring Boot, meaning the core of the application is in Java. MVC is a software development pattern that emphasizes the separation of data representation code from methods that interact with data or process the data. MySQL is used for the relational database.

A screenshot of a cell phone

Description automatically generated

Figure 1 - EER Diagram of database schema

Above is an enhanced entity-relationship (EER) diagram giving a high-level view of the entities and their relationships. These entities/domain classes represent the database tables. These classes are found under **springData.domain**. The **hibernate\_sequence** & **userid\_generator** tables are extra tables generated by Spring Boot to assign IDs to new objects created while using the application.

Spring Boot provides a remember-me cookie implementation which can be enable by adding the **persistent logins** table to the schema. Spring Boot automatically uses the table to store user’s details when they select the remember-me option during login and stores a cookie in the browser.

A controller processes a request and builds an appropriate model and passes it to the view for rendering. Most of the application’s behavior is defined in the controllers found in **springData.controller**. For instance, **AccountController** defines the behavior of and processes requests with “/*account*” such as changing password, viewing profile information, and password reset. Following the MVC pattern allows us to make our system modular and encapsulated. The modularity of the system makes it easier to change parts of the system without having to change other parts of the system.

A close up of a piece of paper

Description automatically generated

Figure 2 - Diagram showing the MVC architecture of the application

**springData.DTO** contains the application’s data-transfer objects (DTOs). The application uses these to collect and display data. DTOs are hollow objects that allow us to move only as much data as is required. DTOs also allow us to translate data from one shape to another such as when using the user enters a pickup location for their request (which uses Google Maps autocomplete search). The application employs DTOs to reduce the amount of data being transferred, improving the application’s speed and performance. An example is when an admin edits a user account, using a DTO, the application only needs to collect the details being changed. Another example is when a user wishes to change their password. Instead of having the application retrieve all the user’s details (and related objects due to the nature of the relational database) then changing the password, the application only needs to collect and store the new password.

Validators in **springData.validator** perform checks on the data received in the DTOs. These ensure that input from the user is in the correct format before the controllers can process the request. I am working on moving validations from the back end to the front-end where possible. These will be implemented using JavaScript and jQuery.

The application also has some helper classes which support the actions of the controllers. **springData.services** implements two services; one checks for and retrieves the details of the user when they attempt to login, the other provides the application’s emailing service. The emailing service implements a third-party library [6] that makes it easy to send emails using custom FreeMarker templates [7].

**springData.utils** implements two utility classes; **AccessCodeGenerator** produces short, alpha-numeric codes provided to users when their requests are fulfilled. The access codes would allow users to enter the cars in a practical implementation of this system. **PasswordGenerator** produces a shuffled, randomly generated password containing numbers, letters, and special characters. The generated password is sent in an email when a user’s password is reset.

The application stores and retrieves data from the relational database using Spring Data JPA. We can contact the database through repositories in **springData.repository** which extend Spring Boot’s JpaRepository class. The JpaRepository class combines the CRUD functionality of the **CrudRepository** and the pagination and sorting of the **PagingAndSortingRepository**. These repositories also allow us to create custom SQL queries to retrieve specific data needed by various controller operations.

The application’s test cases are found under **src/test/**. Unit tests found under **test/java** ensure that the system’s individual components such as the domain classes behave as designed. **test/groovy** contains the unit and integration tests of the controllers and validators which are written in Groovy as required by the Spock framework.

The views are modified from a [Bootstrap template](https://startbootstrap.com/previews/sb-admin-2/). The Front-end uses the following technologies:

* **HTML** – to render the views.
* **CSS/Bootstrap** – to design/style the views.
* **JavaScript/jQuery** – for Timepicker for user requests, client-side validation, Google Maps search autocomplete.
* **Thymeleaf** – sending model data between the controllers and the views. Used to collect/display data alongside DTOs. Also used to send data to the email service templates.

## Algorithms

“*In a car-sharing system, a company offers cars to customers for a period of time. Customers can pick up a car in one location, drive it to another location, and return it there. Car booking requests arrive on-line, and the goal is to maximize the profit obtained from satisfied requests. We refer to this problem as the car-sharing problem*” [1]. In [1], the authors study a car-sharing problem scenario where requests are received continuously (on-line) and go to and from a fixed airport location. In this project, the car-sharing scenario has many cars and many locations, which is similar to what is studied in [11] but differs in that it has no fixed locations.

The algorithm will try to search for a solution where it is desirable to allocate the cars to users. Currently, user requests are served in a first-come first serve (FIFO) basis. If there are no cars available, user requests are simply denied.

I will be implementing the Haversine Formula [8] to calculate the distance between a user’s requested pickup location and a car’s currently parked location (drop-off location of previous request) to find cars within a certain distance. The Haversine Formula will serve as a precursor step for the main car-sharing algorithms.

Ideally, the algorithm will also consider when next a car will be available - hence the **car\_availability** table in the schema. In a scenario where the system receives two requests ***X*** and ***Y*** at 12:00. ***X*** (from point *B* to *A* between 15:10-17:00) and ***Y*** (from *A* to *B* between 13:00-15:00) with one car available. The algorithm should assign the car to both requests at the same time. The current allocation algorithm would satisfy the first request and deny the other.

# Planning and Timescales

## Semester 1

A screenshot of a computer

Description automatically generated

Figure 3 – Partial view of the initial semester 1 Gantt chart (found in docs/1\_plan)

The main goal for semester 1 was to produce a prototype implementing at least two functional requirements and four recommended/quality/optional requirements. This goal was accomplished by implementing:

* **Two** functional requirements **F1** & **F5**.
* **Five** recommended requirements **R1** **- R4** & **R7**.
* **One** quality requirement **Q1**.
* Admin CRUD features as optional requirements.

While implementing the requirements listed above, groundwork for upcoming requirements (**F2**, **F3**, **F4**, **F6** as well as other recommended and quality requirements) was also laid. Given the current state of the software product, most of the intended features have either been fully implemented or have been partially completed.

The plan was mostly followed as intended, although, the experimentation with algorithms (3.4 on the semester 1 Gantt chart) has taken longer than expected. Tasks 4 - 5.3 have been moved to the second semester plan as the Interim Report and Career Plan deadlines were changed. To reflect these changes in the timescales, an updated Gantt chart of the first semester has been added.

A screenshot of a social media post

Description automatically generatedA screenshot of a cell phone

Description automatically generated

Figure 4 - Overview of the updated semester 1 Gantt chart (found in docs/1\_plan)

## Semester 2

The main goal for semester 2 is to quickly complete the implementation of the software features and focus mainly on algorithm design, implementation, and optimization. At the same time, performing unit and integration testing.

* **Interim report** **&** **Career Plan** (13th January - 14th February)
* **Interview** (2nd – 6th March)
* Complete implementation of software system (13th January – 30th April)
  + Algorithm design, experimentation and analysis (13th January – 30th April)
  + Implement remaining requirements (13th January - 31st March)
  + Testing, debugging (31st March - 14th April)
  + Software system evaluation (15th – 19th April)
* **Submit software system & dissertation** (30th April)

**Mini viva** (5th-22nd May)