Computer Science Department

University of Leicester

GreenCar@UoL

CO7201 MSc Individual Project - Dissertation

Taiwo Otubamowo, BSc.

139051137

tao11@student.le.ac.uk

Project supervisor: Rob van Stee, Dr
Second marker: Artur Boronat, Dr

9,431 words
15th May, 2015

Abstract

GreenCar@UoL is a car sharing system for users embarking on University related affairs to save journey cost by sharing their car with other users that embark on similar journeys. Existing car sharing system requires users to manually search for journey peers[1]; GreenCar@UoL on the other hand suggests individual journey peers and combinations of journey peers. Suggesting a combination of journey peers involves computing a feasible path through the origin and destination points of journeys in the combination. Suitable combinations of journey peers clearly offer more cost saving than single occupancy drive or car sharing with one journey peer.

This project formulates the problem of computing a feasible path for a combination of journey peers mathematically as a combination of optimization and precedence constraint satisfaction problems. It also explores a heuristic greedy algorithm for solving the problem given a feasibility upper bound[2].

The software results of this project is a scalable Java based web application to support the car sharing system; a GIS routing component with a map user interface; and an implementation of the heuristic greedy algorithm for computing a feasible path through the origin and destination points of journeys in different combinations of journey peers.

[1] See section 2.2 for definition

[2] The cost of above which a path will be considered impractical; see section 2.2

Declaration

All sentences or passages quoted in this report, or computer code of any form whatsoever used and/or submitted at any stages, which are taken from other people's work have been specifically acknowledged by clear citation of the source, specifying author, work, date and page(s). Any part of my own written work, or software coding, which is substantially based upon other people's work, is duly accompanied by clear citation of the source, specifying author, work, date and page(s). I understand that failure to do these amounts to plagiarism and will be considered grounds for failure in this module and the degree examination as a whole.

Name: Taiwo Otubamowo

Signed: tao11

Date: 15/05/2015

Acknowledgement

I would like to thank my family and friends for their support in my pursuit of a career in computer sciences over the last decade.

I would like to thank my supervisor Dr. Rob for his timely suggestions, trust and encouragement during this project and my second marker Dr. Artur for proposing this interesting project.

Most of all, I would like to give many thanks to God for his gift of life and well-being.

Table of Contents

List of figures

List of tables

1 Introduction

1.1 Background

Car sharing is the practice of sharing a car for a journey so that more than one person travels in the car, reducing travel costs, carbon emissions, traffic congestion and the need for parking spaces (Boronat 2014). GreenCar@UoL is a software system that supports a car sharing scheme for people travelling on affairs to the University of Leicester in order to save journey cost by sharing with other users that embark on similar journeys.

Existing car sharing systems require users to manually search the system for suitable journey peers; while the main functionality of GreenCar@UoL is to suggest suitable suggests suitable journeys, although users ultimately decide to share their car based on the suggestions and personal societal preferences.

1.2 Objectives

The objective of this project is to develop GreenCar@UoL as a fully-fledged software system similar to (Liftshare.com 2015) including the main functionality of suggesting suitable individual journey peers and combinations of journey peers with a sequence of how the driver will pick up and drop off riders for each combination and user interfaces for displaying journey routes on a map.

Journey peer suggestion is central to the system to be developed; it is important that suggestions are computed efficiently because as the number of riders grows large, an exhaustive search of suitable combinations is computationally expensive. The second objective of this project is to design an efficient algorithm that can find suitable combinations in reasonable time for a web based application.

1.3 Motivations and challenges

The motivation for computing combinations of journey peers is that the travel cost of a driver and riders in a combination (two or more riders) is less than the travel cost of single occupancy drive for their individual journeys or car sharing with a single journey peer. Hence, GreenCar@UoL offers opportunities for minimizing journey costs and reducing the number of cars plying roads in Leicester thereby reducing carbon emission and making the environment greener.

GreenCar@UoL is a web based application with a GIS component, designing such system possesses software engineering challenges. GIS data structures are cumbersome and not designed to be used with object-oriented software design. Integrating the GIS component with OOP design is very challenging and requires a good software design to accomplish.

Overall, this project provides an ample opportunity to gain valuable experience in developing scalable enterprise web applications with a GIS component. GreenCar@UoL relies on road data; hence it provides an opportunity to learn how to develop GIS systems for web applications using open source technologies and tools.

1.4 Results

The software implementation of this project is a scalable Java based web application with a GIS component that can be deployed on Heroku cloud application platform. The web application user interface is based on HTML5, CSS3 and JavaScript; it is accessible via a web browser and has a map UI for journey creation. The web application backend is developed using the Java Spring Web MVC framework and JSP, the backend is accessible

via a RESTful JSON API. In addition to the web application is a routing module based on OSM road network data. The routing module is written in SQL for a PostGIS enabled PostgreSQL database; it leverages on functions from the PGRouting library.

The theoretical result is a mathematical formulation of the problem of computing a valid a sequence of how to pickup and drop off riders for a combination of journey peers in a car sharing context. The problem is both an optimization and precedence constraint satisfaction problem. The theoretical result also includes a greedy heuristic algorithm for solving the problem. The algorithm does not always find a valid solution even if one exists. However, it's time complexity is a polynomial of the number of riders in a combination and the output is valid whenever it finds one.

1.5 Outline of subsequent sections

The Literature survey describes the context of car sharing – definitions, types and examples of car sharing systems. It also describes the mathematical formulation of computing a feasible path for a combination of journey peers and the context of relevant technologies used during the implementation of the software component.

The next section describes the software product in terms of its specifications - functional requirements and non-functional requirements; the high level architectural design devoid of platform specific or implementation details; and the product description.

The fourth section is a documentation of the design, pseudo code and tests performed on the algorithm developed to find with feasible paths for combinations of journey peers. It also contains special test cases to demonstrate how the algorithm works.

The next section describes the design, implementation, integration and test of the Java web application, map GUI and database routing components.

The final section concludes the dissertation with a brief overview and appraisal of results achieved; contributions of the project to car sharing; and reflections on significant theories and practical skills learnt on the project.

2 Literature Survey

2.1 Context of car sharing

The principle of car sharing is that a car owner can reduce travel cost by sharing the car and travel cost of journeys with commuters that gain the benefits of private cars without the costs and responsibilities of ownership. Other commuters could be car owners as well but choose not to drive for personal reasons or the economic reason of reducing travel cost by sharing it. Car sharing is most suitable for a community of individuals that have the same objective such as a university, an office or a neighbourhood.

There are different types of car sharing schemes depending on the transportation needs or the type of its users. According to (Taylor Lightfoot Transport Consultants 1995) *"car sharing can principally be divided into three categories:*

1. *Private Car Sharing: Within the family, informally organised, no contracts.*
2. *Neighbourhood Car Sharing: In the neighbourhood, among friends; organised in a simple way, contractual agreements.*
3. *Organised Car Sharing in Professional Car Sharing Organisations: Car sharing organisations administer a centralised/supported network of large-scale user groups and vehicles for shared use".*

GreenCar@UoL falls under a variation of the third category since the university will maintain the platform that supports the scheme and the user group is well defined - individuals travelling on university related affairs.

It is evident that most car owners in cities drive alone thereby car usage is not optimized especially for cases of repetitive journeys such as travel from home to work and vice-versa while other commuters travel the same route around the same time. A scheme that enables commuters to view journeys and identify other commuter for possible peering can cut down peak time travel and on-street parking (Taylor Lightfoot Transport Consultants 1995). A similar scheme is organized by Cranfield University which aims to reduce staff commuting to campus by single occupancy in a car that can accommodate two to four commuters (Cranfield University 2013).

2.2 Computing a feasible path for s combination of journey peers

For a given driver's journey, a journey peer is a journey created by another user with rider modality such that the cost of deviating to pickup and drop off the rider is less than a given factor of the driver's journey cost. It is common place that cars have three passenger seats capacity; hence, car sharing can involve combination of journey peers. Computing a feasible path for all possible combination of journey peers is a hard computational problem. It involves computing an optimal sequence of how the driver should pickup and drop off each rider in the combination such that the overall cost of deviating to pick up and drop off the riders is within the driver's deviation limit. The deviation limit gives an upper bound above which a driver's deviation to pickup rider(s) is considered impractical. If an optimal sequence exist within the deviation limit, then the combination is considered feasible.

In the context of graph theory, an optimal sequence to pickup and drop off riders in a combination is the shortest path through a network of vertices representing the source and destination (referred to as sink hereafter) points of each rider in the combination with the following constraints:

- A rider's source vertex must precede its sink vertex in the sequence
- All riders' sink vertices must precede the driver's sink vertex in the sequence

The problem can be formulated with the parameters used in formulating the travelling sales man problem in (Garey and Johnson 1979) as follows:

Given a driver and a set of riders $R = \{r_1, r_2, \dots, r_n\}$; the problem is a set of vertices $V = \{v_0, v_1, v_2, \dots, v_n, v_{n+1}, v_{n+2}, \dots, v_{2n}, v_{2n+1}\}$ where v_i and v_{i+n} are the source and sink vertices of rider r_i respectively; v_0 and v_{2n+1} are the source and sink vertices of the driver respectively; $h(v_i, v_j)$ is the distance between v_i and v_j.

The solution is an ordering $S = \langle v_0, v_{\pi(1)}, v_{\pi(2)}, \dots, v_{\pi(2n)}, v_{2n+1} \rangle$ of V that minimizes $h(v_0, v_{\pi(1)}) + \left[\sum_{i=1}^{2n-1} h(v_{\pi(i)}, v_{\pi(i+1)})\right] + h(v_{\pi(2n)}, v_{2n+1})$ and satisfies the precedence constraint $v_i \prec v_{i+n};\ v_i, v_{i+n} \in S\ \forall\ r_i \in R$.

The optimality expression gives the cost of the ordering where the driver traverse her source, then the riders' source and sink vertices according to the ordering and then traverses her sink.

The precedence constraint can be checked by the inequality $cost(v_0 \rightarrow v_i) < cost(v_0 \rightarrow v_{i+n})$ in the ordering $S\ \forall\ r_i$. The ordering S is a Hamiltonian path as every vertex is visited exactly once in the ordering.

The problem has an optimization component and a precedence constraint satisfaction component. Similar problem are the well-known travelling sales man problem (nature of input and optimization component) and the Sequential ordering problem (precedence constraint satisfaction component).

2.3 Related work and relevant future research

Computing feasible combination of journey peers is a hard problem. In this project, a heuristic greedy algorithm design was used to solve the problem. Other research could explore meta-heuristic or exact algorithm design as described in (Ropke 2005) to solve the problem with a comparison of results. Such research will provide valuable conclusion on the preferable algorithm for solving the problem. The main metrics for deciding a good algorithm for the problem are how often an algorithm provides an optimal solution and the variance of solution costs to the optimal solution cost.

Computer science research on optimization and constraint satisfaction problems is pertinent to designing suitable algorithms for solving the problem of finding feasible combinations of journey peers that have feasible paths. The vehicle routing problem with time windows (VRPTW) is one of such optimization and constraint satisfaction problems similar to the computational problem in this project. (Pisinger and Ropke 2007) explores popular heuristic algorithms that have been applied to the VRTPW.

2.4 Context of implementation technologies

This section describes noteworthy technologies (tools, languages, frameworks and methodologies) used during the implementation of the software system on this project.

2.4.1 GIS

GIS, Geographical Information System is a system of hardware and software for collecting, managing, analysing and displaying real world temporal information (attributes) that are related to the spatial feature of the earth. Data in GIS are traditionally represented as a layer which can be superimposed on another layer containing the spatial information of the earth in terms of positions with reference to latitude and longitude.

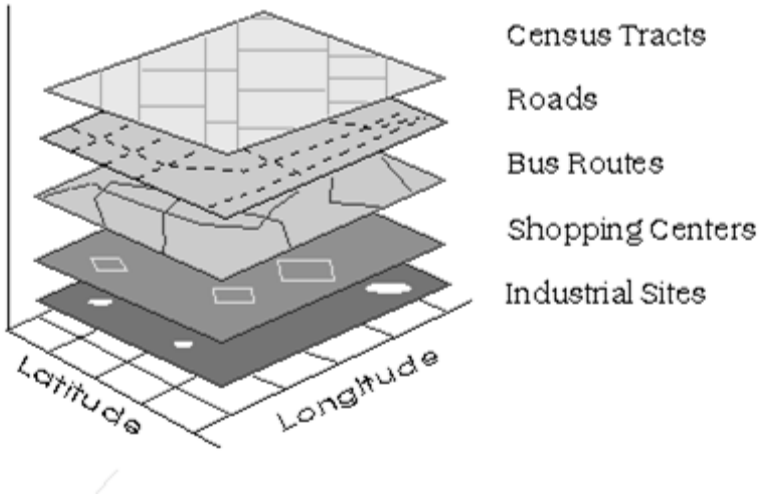

Figure 1: Layers in GIS (Foote and Lynch 2014)

Data in GIS are either temporal or spatial; temporal data is referenced to date/time of occurrence while spatial is reference to latitude and longitude. Spatial data can be represented as Vectors or Raster. Vectors are GIS data structures that can be represented as mathematical equation. It includes: Points - a coordinate (x, y) usually latitude and longitude; Line Strings - connects two points, analogous to graph edges; Polygons - three or more closed line strings. Vector based GIS are useful for analysing measurable quantities such as roads networks (length of road is important in computing shortest path routes).

In a Raster based GIS, data structure objects are based on rows and columns of image pixels; each pixel represents a unit of geometric information.

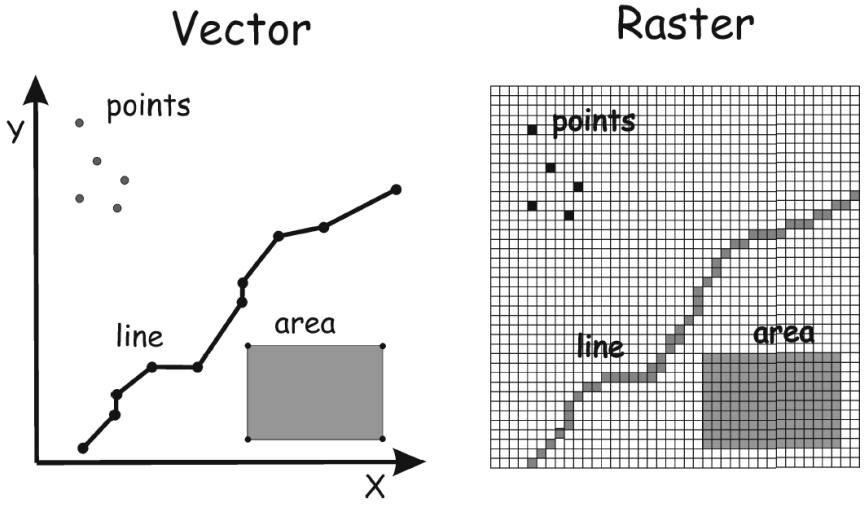

Figure 2: Vector and Raster data model Unable to format

2.4.2 PostgreSQL and PostGIS

PostgreSQL is an open source object relational database management system which allows user to add extensions to its base data types and functionalities. Regular database types are textual (characters or numbers) for storing names, currency and numbers, special type systems are required for representing geometric quantities in a database. PostGIS is a popular

open source geospatial extension for PostgreSQL. PostGIS provides data types, queries and functions optimized for spatial (geometric) data and operations such as comparing two points and finding the closest geographical feature to a point. PostgreSQL and PostGIS combined can provide persistence and GIS functionalities for a web application software implementation.

2.4.3 OpenStreetMap (OSM)

OSM is an open source project that manages crowd sourced map data of the world. Planet.osm contains all data that make up OSM maps. An uncompressed XML variant is over 554GB (Wiki.openstreetmap.org 2015). OSM data is crowd-sourced, hence daily updates are commonplace. A good GIS application system has to cater for such updates for maintain accurate data.

The OSM shapefiles vector based data format is used in this project. It contains buildings, points, railways and roads among others as XML documents. It can be imported into spatially enabled database (example is a combination of PostgreSQL and PostGIS). OSM is the underlying data of the system. Google Maps and Bing Maps also provide free map data, however, they have usage limit constraints and not open source.

2.4.4 GeoServer and OpenLayers

GeoServer is a Java-based map server with inbuilt Jetty application server that allows users to view and edit geospatial data using open standards set forth by Open Geospatial Consortium (OGC). GeoServer allows for great flexibility in map creation and data sharing. Jetty is an open source Java Servlet Engine and HTTP web server. GeoServer provides Web Map Service (WMS) for map rendering map images over HTTP. WMS is an OGC specification that provides a simple HTTP interface for requesting geo-registered map images from one or more distributed geospatial databases (Open Geospatial Consortium Inc. 2006). GeoServer and OpenLayers are part of the routing component in this project (see section 5.2).

OpenLayers is an open source JavaScript library to load, display and render maps from multiple sources on web pages. It leverages the functionalities of HTML5 like the canvas tag. OpenLayers enables an intuitive way of communicating with a WMS from JavaScript.

2.4.5 Spring Web MVC

MCV stands for Model View Controller architecture. It is a guideline for developing an application having a user interface. In terms of Java web applications; an application written with the MVC framework consists of:

1. Models consist of data used by the application and business logic for accessing the data used by the application. Models are written as Java POJOs, Entities and Data Access Objects.
2. Views consist of several JSP pages (including HTML5 and JavaScript) that handle the UI logic only.
3. Controllers handle application flow and logic. They handle HTTP request from clients, determine which model to fetch and a suitable view to present the fetched model. The model and the view is then bound and sent as a HTTP response to clients. Controllers are coded in Java Servlets.

This architecture allows separation of concern as massive changes can be made to individual components of the architecture without changing the other components.

Spring MVC is a JavaEE web framework based on the MVC architecture. Spring MVC takes care of web application components (security, access control, database connection management and logging) that are similar across disparate system via a powerful annotation based configuration system and dependency injection container (also known as inversion of control).

3 Requirements specification, system architecture and product description

This section describes the requirements specification, high level software architecture (no implementation details) and a detailed description of the product (GreenCar@UoL).

3.1 Requirements specification

The following section is a detailed specification of the software system in this project. The requirements are listed as functional, non-functional requirements and a use case diagram that describes the interaction the user will have with the system.

3.4.1 Functional requirements

1 System must support multiple users via registered individual `User` accounts.

2 A user should be able to login to the system using a registered `User` account and logout after a successful login.

3 A `Journey` is a unit of data that has two points (origin and destination), modality (driver or rider), purpose, time of departure, frequency (daily, weekly, occasional, one-off and shift), and optional comments.

4 A journey is associated with one and only one existing `User` account.

5 An authenticated user can create many journeys.

6 All users can view all journeys (authenticated or anonymous).

7 An authenticated user can update and delete journeys associated with her `User` account.

8 A user cannot update or delete journeys associated with another `User` account.

9 For a given journey with driver modality associated with a `User` account (driver), a journey peer is a journey with rider modality associated with a different User account (rider) such that the cost of deviating to pick up and drop off the rider is less than the driver's deviation limit.

10 An authenticated user can view journey peers for journeys associated with another `User` account with driver modality.

11 An authenticated user can view suitable combinations of journey peers such that the cost of deviating to pickup and drop off every rider in the combination is less than the driver's deviation limit.

12 A suitable combination of journey peers should have a path from the driver's origin to its destination where each rider in the combination is picked up and dropped off.

3.4.2 Non Functional requirements

1 Journeys and user accounts detail are stored to and retrieved from a web-based application that is deployable to the cloud.

2 The system must include a genuine, flexible, cost-efficient mechanism to compute routes for journeys that is trustworthy. Data from OpenStreetMap should be used to compute journey routes.

3 The web application client UI should consist of views to manage user accounts (including user preferences), journey requests and statistics.

4 Performance and scalability are important aspects of the system. In particular: the response time of the system should be acceptable in order to offer an appealing user experience; the system should be available at all times; and it should be able to deal with the whole planet map from OpenStreetMap.

5 Finding suitable combinations of journey peer could be time consuming depending on input size and server resources; this condition requires a communication model that allows the server to send results to client long after a request was made. Hence,

traditional request/response model of the HTTP protocol is not suitable for this system. This project will employ WebSocket for communication, WebSocket allows a bi-directional message exchange between a client and a server; requests can be made by either the client or server after an initial HTTP handshake (Park *et al.* 2014).

6 Access control must be clearly defined to demarcate operations available to authenticated users only and operations available to all users.

7 Web component of the system should expose a RESTful JSON API for operations available to users (to facilitate with a mobile application user interface).

8 An email notification system that is triggered whenever a new receives a journey peering request or a new journey that can be peered with the user's journey is created.

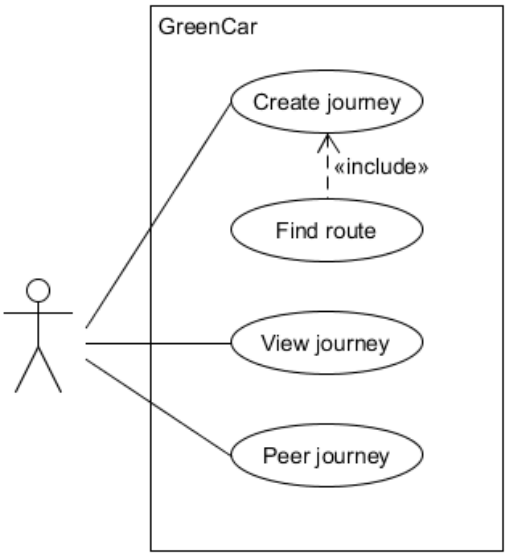

Figure 3: Main user interactions with system caputered with use cases

3.1 High level architectural and design

The high level architecture shows the connection between the system components (layers) with respect to the system requirements specification. The system Application client (user interface) consists of web pages based on HTML5, JavaScript and CSS3. The main pages include the home page, login page, journey view, edit and create pages and journey peer page.

The server is a Java based web application that contains the application logic (security, access control, API) and connects to the database layer.

The database layer consists of tables (journeys and users) for persistent storage of system data.

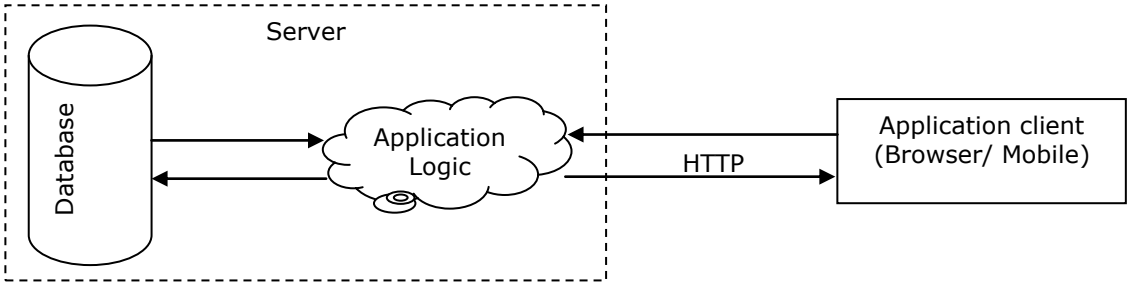

Figure 4: High level overview of system architecture

3.4.3 Data model

The class diagrams below captures the structural view of the design of the system using the model-based approach in (Heckel and Lohmann 2003). It is important to note that `Journey` and `User` are persistent entities; hence, they have corresponding tables in the database while `RiderJourney, JourneyPeer, Combination, Point` and `Route` are used only for representing objects during runtime – no persistence.

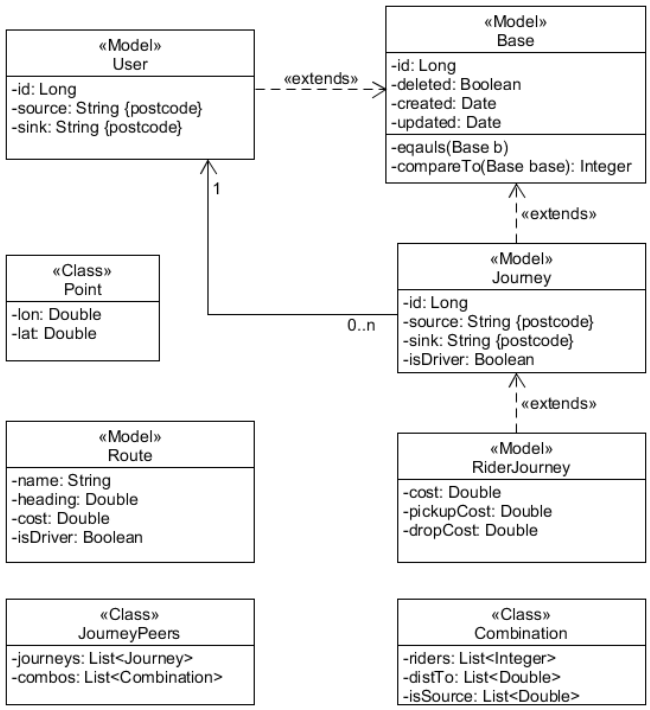

Figure 5: Data model of system

3.2 Product description

This section details the description of the product developed based on the system's requirements specification, high level architecture and data model.

The purpose of the product (GreenCar@UoL) is to support car sharing for people travelling on affairs related to the University of Leicester. The primary users are staff and student of the university. It provides journey peer suggestions that provide journey cost savings when they are followed. The product is available on the web through an internet-connected web browser. The application can support any size of users as it is cloud based (scales both upwards and downwards).

The product is composed of the user interface; cloud based web application and a database. Section 5 contains a full description of how the product is designed.

4 Journey peer algorithm design

The main challenge of this project is designing an algorithm to compute feasible combination of journey peers. This section is dedicated to describing the design of the algorithm and test cases that demonstrate scenarios where the algorithm: finds an optimal solution; finds solutions that are not optimal; and does not find any solution at all when one clearly exists.

4.1 Exhaustive search

Finding all combination of riders is trivial as it can be accomplished by computing the power set of the set of riders $r = \{1,2,\cdots,n\}$. On the other hand, finding a valid sequence for each combination is a computationally hard – a combination of an optimization and constraint satisfaction problem. However, it is brute force solvable as shown in the algorithm below:

$foreach(subset\ sr\ in\ power\ set\ of\ r: |sr| > 1)\{$
$\quad perms = \ all\ permutations\ of\ sr$
$\quad foreach(perm\ in\ perms)\{$
$\quad\quad if(perm\ fails\ precedece\ test\ or\ cost(perm) > ref)$
$\quad\quad\quad eliminate\ perm\ from\ perms$
$\quad optSequence = perm\ with\ minimum\ cost$
$\}$

The time complexity of the brute force search described above is at least $O(n!)$ which makes it undesirable for implementation. An implication of the problem being brute force solvable is that the problem belongs to the class of non-deterministic problems - NP.

4.2 Greedy and heuristic approach

This explain this approach, it is pertinent to define the following:

$R = \{1,2,\cdots,n\}$; A set of consecutive integers from 1 to n individual journey peers

$s = 0;$ source vertex of the driver

$t = 2n - 1;$ sink vertex of the driver

$h(v,w)$; length of shortest $v - w$ path

$ref = \frac{4}{3} * h(s,t)$; reference cost / feasibility upper bound / deviation limit

$G;$ A weighted digraph with $V = \{0,1,2,\ldots,n,n{+}1,n{+}2,\ldots,2n,2n{+}1\}$

$m = |V| = 2n + 2.$

The reference cost could be any constant set by the driver; however for simplicity it is safe to assume that a driver's deviation limit is four-third the cost of her original journey. The rationale being that if the cost of the new journey is split in half between the driver and the rider, the driver will have a cost two-third of her original journey.

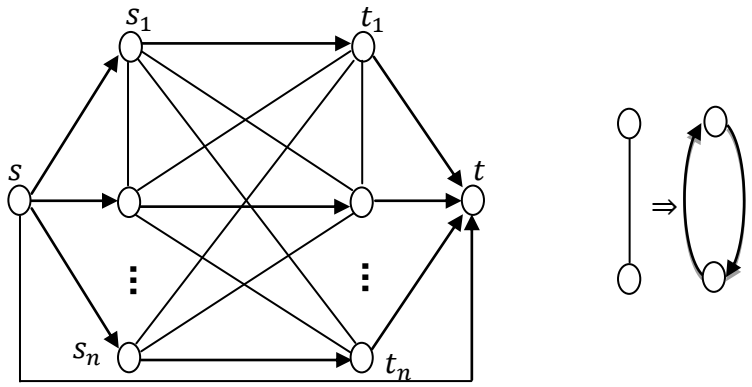

Figure 6: Network of driver and riders source and sink vertices

The network in Figure 7 represents graph G, an undirected edge between two vertices implies there are two directed edges in opposite between the vertices each with separate cost which could be disparate. There is a directed edge from:

- the driver's source to all the riders source $(s, i) \forall i \in R$;
- every rider's source to the corresponding sink $(i, i{+}n) \forall i \in R$;
- every rider's source to other rider's source and sink $(i, j), (i, j{+}n) \forall i, j \in R, i \neq j$;
- every rider's sink to other rider's source and sink $(i{+}n, j), (i{+}n, j{+}n) \forall i, j \in R, i \neq j$;
- every rider's sink to the driver's sink $(i, t) \forall i \in R$.

The edges are selected carefully to avoid wasting resources in computing costs of edges that cannot be part of any valid solution. Computing the cost of an edge requires computing the shortest path between the vertices connected by that edge. Using Dijkstra's algorithm implemented with min heap, the time complexity of computing the cost of edges in G is $O(|E^*| \log |V^*|)|E| \; where \; |E| = 1 + 2m(2m-1)$ and E^*, V^* are the set of edges and vertices in the underlying OSM road network.

The main component of the heuristic greedy algorithm is a routine $getPath(v)$ that traverses a vertex v and recursively calls $getPath(w)$ where the vertex w is connected to the minimum cost edge adjacent to v. The routine is started by running it on the driver's source. It terminates when the total cost exceeds ref or it traverse t.

A valid sequence is a path $p = s - t \; path : length(p) < ref \; and \; s_i \; precedes \; t_n$.The solution can be represented as a string of length that is polynomial in the input size of the problem – precisely the output length is m. Any purported solution can be verified in polynomial time by checking if i precede $i + n$ in the solution for all $i \in R$, hence the proof that the problem is in NP.

4.2 Pseudo code

A smart numbering of the vertices allow for easy access to the source and sink nodes of the driver and riders in the graph. For a graph with n riders, the source and sink vertex of rider i is labelled i and $i + n$ respectively. The driver's source and sink vertices are labelled 0 and $V - 1$ respectively.

Below is the pseudo-code of a heuristic greedy algorithm developed to solve the computation problem. The core of the algorithm is a recursive call to $expand(v)$; it starts with $expand(s)$, finds the vertex v connected to the least cost adjacent edge from s then

computes $expand(v)$ recursively until the path connects the vertex t or the cost of the edge exceeds ref. The heuristic is based on a conjecture that a feasible path will include all the riders in a combination so it always seem better to expand the vertex connected to the least cost edge (ties are broken arbitrarily) from the current vertex being expanded. The algorithm addresses the precedence constraint by deactivating the riders' sink vertices until the corresponding source vertex is expanded and it makes t expandable when all riders' sink vertices have been expanded.

```
inputs:
n − number of riders
G − edge weighted digraph with vertices labelled with indices 0,1,2,⋯,V − 1;
where vertices labelled 1,⋯,n represent riders 1,⋯,n source respectively and vertices
labelled n+1,⋯,n+n represents riders 1,⋯,n sink respectively and V is the number of
vertices in G.
computeCombinations(n, G){
    s = 0, t = V − 1;
    r = [1,2,⋯,n]
    foreach(subset sr in poswer set of r : |sr| > 1){
        initialize arrays activated, distTo, nextVertex, visited each of size V
        activated[s] = true;
        foreach(s_i in sr) activated[s_i] = true;
        distTo[s] = 0.00;
        expand(s);
        if(visited[t]){
            v = s;
            while(v ≠ t){
                w = nextvertex[v];
                print(v → w, distTo[w])
                v = w;
            }
        }
    }
}

expand(v){
 visited[v] = true;
        if(visited[s_i + n] = true ∀ s_i in sr) activated[t] = true;
        initialize minPQ a key-value minHeap
        foreach(edge in G.adj(v)){
            if(activated[edge.to] = true & visited[edge.to] = false)
                minPQ.insert(edge.to, edge.wieight);
        }
        if(minPQ is not empty){
            w = minPQ.minIndex; ties are broken arbitrarily
            cost = minPQ.minValue;
            minPQ.del(w);
            if(1 ≤ w ≤ n)activated[w + n] = true;
            nextVertex[v] = w;
            distTo[w] = distTo[v] + cost;
            if(distTo[w] < ref)getPath(w);
```

```
        }
}
```

4.3 Testing

A Java based demo implementation of the algorithm can be accessed on the web with the link http://greencar.herokuapp.com/api/demos/peer/{ref}/{n} where ref is the reference cost and n is the number of rider. The demo generates random edge weights and uses the algorithm described above to compute valid sequence for all possible combinations of riders. The output is a graph with the randomly generated edge weights and feasible combinations with the path (ordering of vertices) and cost.

The behaviour of the algorithm can be traced using desk check by monitoring the state of the variable used in the algorithm for each recursive call to $expand(v)$.

Test case 1

Sample input:

$refCost = 19.99;\ n = 3$

0: 0 → 3 0.10 0 → 2 0.10 0 → 1 0.10
1: 1 → 6 4.20 1 → 3 0.90 1 → 5 1.00 1 → 2 0.80 1 → 4 0.80
2: 2 → 6 4.80 2 → 3 4.80 2 → 4 0.10 2 → 1 0.10 2 → 5 8.00
3: 3 → 5 3.00 3 → 2 1.80 3 → 4 6.00 3 → 1 1.80 3 → 6 12.60
4: 4 → 6 0.10 4 → 3 8.40 4 → 5 16.00 4 → 2 4.00 4 → 7 8.40
5: 5 → 6 6.00 5 → 3 7.50 5 → 4 0.10 5 → 1 0.10 5 → 7 2.00
6: 6 → 5 0.10 6 → 2 1.20 6 → 4 4.80 6 → 1 1.80 6 → 7 14.00
7:

Output:

[2, 3]
0 → 3 (0.10) 3 → 2 (1.90) 2 → 6 (6.70) 6 → 5 (6.80) 5 → 7 (8.80)
[1, 3]
0 → 3 (0.10) 3 → 1 (1.90) 1 → 4 (2.70) 4 → 6 (2.80) 6 → 7 (16.80)
[1, 2]
0 → 2 (0.10) 2 → 1 (0.20) 1 → 4 (1.00) 4 → 5 (17.00) 5 → 7 (19.00)
[1, 2, 3]
0 → 3 (0.10) 3 → 2 (1.90) 2 → 1 (2.00) 1 → 4 (2.80) 4 → 6 (2.90) 6 → 5 (3.00) 5 → 7 (5.00)

The table below is the result of the desk check test for riders {2,3} - a subset of the set of riders {1,2,3} using the sample input described in test case 1.

v	$activated[]$	$visited[]$	$nextvertex[]$	$distTo[]$
-	$[t,f,t,t,f,f,f,f]$	$[f,f,f,f,f,f,f,f]$	[−, −, −, −, −, −, −, −]	[0.0, −, −, −, −, −, −, −]
0	$[t,f,t,t,f,f,t,f]$	$[t,f,f,f,f,f,f,f]$	[3, −, −, −, −, −, −, −]	[0.0, −, −,0.1, −, −, −, −]
3	$[t,f,t,t,f,t,t,f]$	$[t,f,f,t,f,f,f,f]$	[3, −, −,2, −, −, −, −]	[0.0, −,1.9,0.1, −, −, −, −]
2	$[t,f,t,t,f,t,t,f]$	$[t,f,t,t,f,f,f,f]$	[3, −,6,2, −, −, −, −]	[0.0, −,1.9,0.1, −,6.7, −]
6	$[t,f,t,t,f,t,t,f]$	$[t,f,t,t,f,f,t,f]$	[3, −,6,2, −, −,5, −]	[0.0, −,1.9,0.1, −,6.8,6.7, −]
5	$[t,f,t,t,f,t,t,f]$	$[t,f,t,t,f,t,t,f]$	[3, −,6,2, −,7,5, −]	[0.0, −,1.9,0.1, −,6.8,6.7,8.8]
7	$[t,f,t,t,f,t,t,t]$	$[t,f,t,t,f,t,t,t]$	[3, −,6,2, −,7,5, −]	[0.0, −,1.9,0.1, −,6.8,6.7,8.8]

Table 1: Desk check of journey peer algorithm

The table below is the result of the desk check test for riders {1,2,3} - a subset of the set of riders {1,2,3} using the input described in test case 1.

v	$activated[]$	$visited[]$	$nextvertex[]$	$distTo[]$
-	$[t,t,t,t,f,f,f,f]$	$[f,f,f,f,f,f,f,f]$	$[-,-,-,-,-,-,-,-]$	$[0.0,-,-,-,-,-,-,-]$
0	$[t,t,t,t,f,f,t,f]$	$[t,f,f,f,f,f,f,f]$	$[3,-,-,-,-,-,-,-]$	$[0.0,-,-,0.1,-,-,-,-]$
3	$[t,t,t,t,f,t,t,f]$	$[t,f,f,t,f,f,f,f]$	$[3,-,-,2,-,-,-,-]$	$[0.0,-,1.9,0.1,-,-,-,-]$
2	$[t,t,t,t,t,t,t,f]$	$[t,f,t,t,f,f,f,f]$	$[3,-,1,2,-,-,-,-]$	$[0.0,2.0,1.9,.0.1,-,-,-,-]$
1	$[t,t,t,t,t,t,t,f]$	$[t,t,t,t,f,f,f,f]$	$[3,4,1,2,-,-,-,-]$	$[0.0,2.0,1.9,0.1,2.8,-,-,-]$
4	$[t,t,t,t,t,t,t,f]$	$[t,t,t,t,t,f,f,f]$	$[3,4,1,2,6,-,-,-]$	$[0.0,2.0,1.9,0.1,2.8,-,2.9,-]$
6	$[t,t,t,t,t,t,t,f]$	$[t,t,t,t,t,f,t,f]$	$[3,4,1,2,6,-,5,-]$	$[0.0,2.0,1.9,0.1,2.8,-,2.9,-]$
5	$[t,t,t,t,t,t,t,f]$	$[t,t,t,t,t,t,t,f]$	$[3,4,1,2,6,7,5,-]$	$[0.0,2.0,1.9,0.1,2.8,3.0,2.9,-]$
7	$[t,t,t,t,t,t,t,t]$	$[t,t,t,t,t,t,t,t]$	$[3,4,1,2,6,7,5,-]$	$[0.0,2.0,1.9,0.1,2.8,3.0,2.9,5.0]$

Table 2: Desk check of journey peer algorithm

Analysis

The result of the desk check tests above shows that the behaviour of the algorithm is as expected. It is a greedy algorithm as it commits to the least cost adjacent edge at every vertex. Although the algorithm is recursive, it is guaranteed to terminate because the distance will eventually grow larger than ref in a finite number of steps. When the call to method $expand(s)$ returns, if $visited[t] = true$ implies a valid sequence was found with $cost \leq ref$.
The algorithm finds a feasible sequence or path for some but not all combination of riders with $cost \leq ref$. However, the sequence found by the algorithm for a combination of riders is not always the optimal feasible sequence for that combination of riders.

4.4 Special test cases

This section explores special test cases based on input. It shows that the algorithm does not always find a solution when one clearly exists and solutions found are not always the optimal feasible path for a carefully crafted input.

Test case 2

Sample input:

$refCost = 9.99;\ n = 2$
0: $0 \to 2\ \ 0.10\ \ 0 \to 1\ \ 0.10$
1: $1 \to 4\ \ 2.00\ \ 1 \to 2\ \ 0.10\ \ 1 \to 3\ \ 0.30$
2: $2 \to 3\ \ 4.20\ \ 2 \to 1\ \ 0.10\ \ 2 \to 4\ \ 0.10$
3: $3 \to 4\ \ 0.10\ \ 3 \to 2\ \ 0.10\ \ 3 \to 5\ \ 4.50$
4: $4 \to 3\ \ 6.00\ \ 4 \to 1\ \ 1.20\ \ 4 \to 5\ \ 0.10$
5:

Output:

$[1, 2]$
$0 \to 2\ (0.10)\ \ 2 \to 1\ (0.20)\ \ 1 \to 3\ (0.50)\ \ 3 \to 4\ (0.60)\ \ 4 \to 5\ (0.70)$

Analysis:

The algorithm finds a feasible path which is clearly the optimal feasible path. It greedy choices end up being the optimal solution and it finds it blazingly fast compared to the brute force search.

Test case 3

Sample input:

$refCost = 15.99$

0: 0 → 2 0.10 0 → 1 0.10
1: 1 → 3 7.00 1 → 2 5.00
2: 2 → 4 0.10 2 → 3 10.00 2 → 1 5.00
3: 3 → 4 7.00 3 → 2 10.00 3 → 5 0.10
4: 4 → 3 7.00 4 → 5 0.10
5:

Output:

null

Analysis:

The algorithm terminates when it traverse vertex 4 as the cost at the vertex is greater that the reference cost 0 → 2 (0.1) 2 → 1 (5.1)1 → 3 (12.1) 3 → 4 (19.2). However, the path 0 → 1 (0.1) 1 → 2 (5.1)2 → 4 (5.2) 4 → 3 (12.2)3 → 5 (12.3) is a valid sequence and is indeed the shortest path from 0 → 5. The algorithm fails in this instance

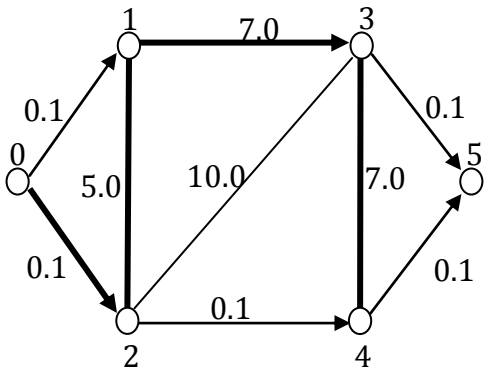

Figure 7: Path through network of a driver and riders' source and sink vertices

5 Implementation

The main software component of GreenCar@UoL is a Java web application including a map GUI that can be deployed to a local environment and to a remote cloud platform – Heroku. The other component of the software is the routing component consisting of a couple of SQL functions and OSM data stored in database tables.
This section includes description of the software implementation in two sub-sections; the first sub-section describes the implementation requirements and plan; the fourth sub-section describes how the components are integrated; and the last sub-section documents the tests performed.

5.1 Implementation plan and technology requirements

The plan during the software development is described in the table below as weekly deliverables.

Time		Work plan / Deliverable
Month 1	Week 1	Overall research of available technologies
	Week 2	Map GUI and routing component research and design
	Week 3	Map GUI implementation and testing (based on OpenLayers)
	Week 4	Routing component implementation (includes PostgreSQL database setup – including PostGIS and PGRouting) Routing component testing Integration of map GUI with routing component using GeoServer.
Month 2	Week 1	Web application component design and implementation
	Week 2	Web application component testing and integration with existing system.
	Week 3	Integration bug fixes
	Week 4	Journey peer algorithm research and design
Month 3	Week 1	Journey peer algorithm research and design
	Week 2	Journey peer algorithm implementation, testing and integration with existing system.
	Week 3	Journey peer integration bug fixes
	Week 4	Code refactoring and GUI optimization
Month 4	Week 1	Code refactoring and GUI optimization
	Week 2	Improvements to journey peer integration and bug fixes
	Week 3	
	Week 4	

Table 3: Weekly implementation deliverables

The implementation has a number of technology requirements or dependencies. The table below lists the technology requirements and reasons for choosing them over close alternative.

Technology	Role	Version	Alternative	Justification
Java	Programming language	1.6	PHP	Provides robust data structures that are more powerful than data structures of other

				programming languages.
Spring Web MVC	It is a Java based web framework that provides MVC architecture and components that can be used to develop loosely coupled web applications.	3.2.4	Play	Spring is a mature framework and its user base is larger than any other Java web framework. Hence, it has a better set of online resources and information.
Hibernate	It is a Java persistence framework popular for proving Object Relational Mapping (ORM) functionality.	5.0.1	iBatis	iBatis provides more control over SQL queries - a feature not needed on this project and Spring MVC supports Hibernate out of the box.
Maven	Project / source code dependency management and build system tool	4.0.0	Gradle	Although Maven is more verbose than Gradle; It is based on XML similar to Spring configuration files providing uniformity in project configuration language.
JUnit	Java unit testing framework	4.11	TestNG	JUnit is a mature test framework and its user base is larger than any other Java test framework. Hence, it has a better set of online resources and information.
PostgreSQL	Relational database server	9.4	MySQL	In contrast to MySQL, PostgreSQL can be easily extended to provide advanced functionalities such as handling geo-spatial data.
PostGIS	Geospatial extension for PostgreSQL relational database	2.1.6	-	
GeoServer	Platform for publishing spatial data to the web.	2.6.2	Mapserver	There is no clear advantage over Mapserver. However, GeoServer runs on Jetty – which can also run Spring Web MVC applications.
OpenLayers	JavaScript library for rendering maps on web pages	3	Leaflet	OpenLayers supports a wider range of formats and GIS functionalities. It also provides direct connectivity with GeoServer.

Table 4: Technology requirements audit

5.2 Routing component

As mentioned earlier, the routing component is wholly implemented on the database layer using OSM data, PostgreSQL, PostGIS and PGRouting. The routing module consists of two tables shown in the image below and a couple of SQL functions.

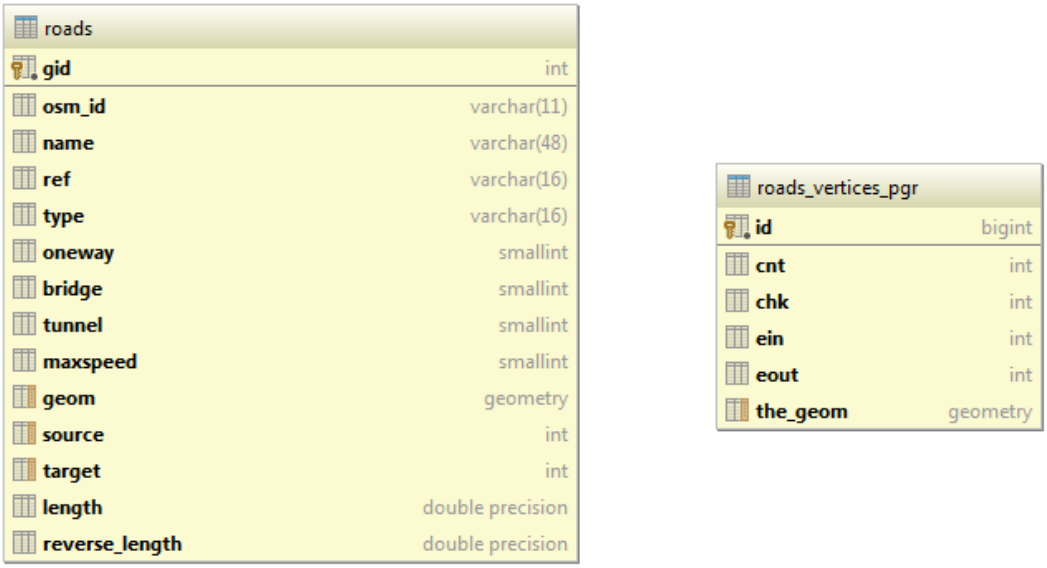

Figure 8: Database tables involved in routing

The roads table is created by importing OSM shapefiles for roads using the shapefiles loader. At the core of the routing module is the PGRouting library which can be used to compute shortest path route for OSM road networks on a PostGIS enabled PostgreSQL databases. See Appendix 1 for detailed steps of how to prepare the roads table for routing. The routing implementation is a wrapper for pgr_dijkstra function (see documentation at http://docs.pgrouting.org/dev/src/dijkstra/doc/index.html). The wrapper modifies the result of the function by excluding all road links that are not accessible by a car as shown below.

```
1. SELECT gid, geom, name, cost, source, target FROM
2.  pgr_dijkstra('SELECT gid AS id, source::int, target::int,
3.  length::float AS cost, reverse_length::float AS reverse_cost FROM '
4.  roads WHERE type IN ("motorway", "motorway_link",
5.  "trunk", "trunk_link", "primary", "primary_link", "secondary",
6.  "secondary_link", "tertiary", "tertiary_link", "unclassified",
7.  "residential", "living_street", "service","track")',
8.  source, target, true, true),
9. roads WHERE id2 = gid ORDER BY seq';
```

Table 5: Code extracts from gcar_dijstra_directed.sql

5.3 Web application component

The web application component is a Java web application based on the Spring Web MVC framework. The application logic provides access control to control actions performed by users as stipulated in functional requirements 2 – 8 as shown in the figure below. The RESTful API controllers in the application provide actions corresponding to the methods in the controllers classes.

```
1.      // Login View
2.      public static final String LOGIN_VIEW = "login";
3.
4.      /**
5.       * Journey Base View Mapping
6.       */
7.      public static final String JOURNEY_BASE_MAPPING = "/journey";
8.      // Sub View Mappings
9.      public static final String JOURNEY_EDIT = "edit"; // GET, POST
10.     public static final String JOURNEY_VIEW = "view"; // GET
11.     public static final String JOURNEY_LIST = "list"; // GET
12.     public static final String JOURNEY_DELETE = "delete";
13.     public static final String JOURNEY_PEER_SINGLE = "peer/single"; // GET
14.     public static final String JOURNEY_PEER_LIST = "peer/list"; // GET
```

Figure 9: Code extract from RouteConfig.java (URL configurations)

5.3.1 The models (entities)

The journeys and users tables in the database are mapped to the Journey and User entity classes (Plain Old Java Objects) respectively using the Hibernate ORM technology. Besides providing persistence for the models, the database layer also house the routing module as described earlier which makes PostgreSQL an excellent choice for persistence. The persistence component of the database can be hot swapped with any other relational database.

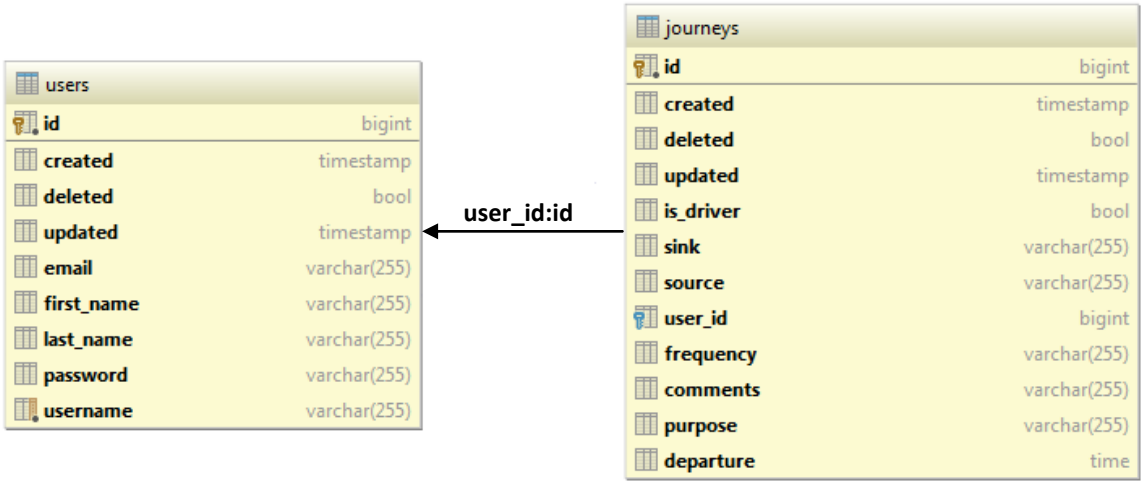

Figure 10: Database tables mapped to models

5.3.2 The controllers

Controllers expose RESTful API for the web application actions over HTTP. Each method in a controller class represents a single action that fetch or change the state of one or more data model described above. The `JourneyController` provides an API for performing CRUD operation on the `Journey` model as the `UserController` is for the `User` model. The RouteController on the other hand exposes an interface for computing the routes between

two points. The figure below shows a class diagram representation of controllers and components in the web application components.

Components do not expose their action over HTTP; hence they are not accessible directly by via the API. However, they are used by the controller classes as helpers for operation that are not specific to any controller class. The `GeoToPostcode` component converts postcodes to latitude and longitude and vice-versa. `PeerCombination` component houses the algorithm for computing the sequence of pickup and drop off of a combination of journey peers.

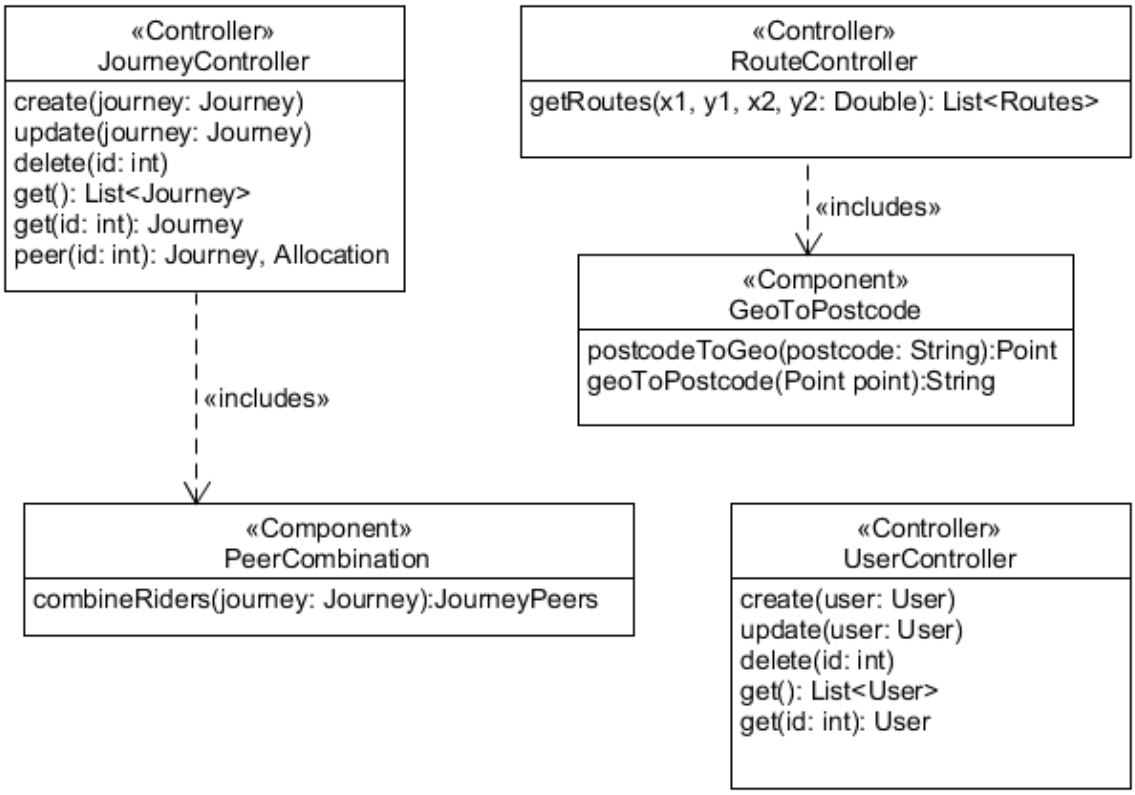

Figure 11: Platform independent class diagram of controllers and components.

5.3.3 The views - GUI

The GUI consists of web pages implemented with CSS, HTML5 and JavaScript. The GUI can be viewed online at http://greencarherokuapp.com. The most significant component of the GUI implementation is in `journey.js` – It contains the logic for requesting the route between two points on the map in `journey/view.jsp` and `journey/edit.jsp`. Below are screen shot of the web pages that represent the GUI, the functionalities of the web pages are described in corresponding captions.

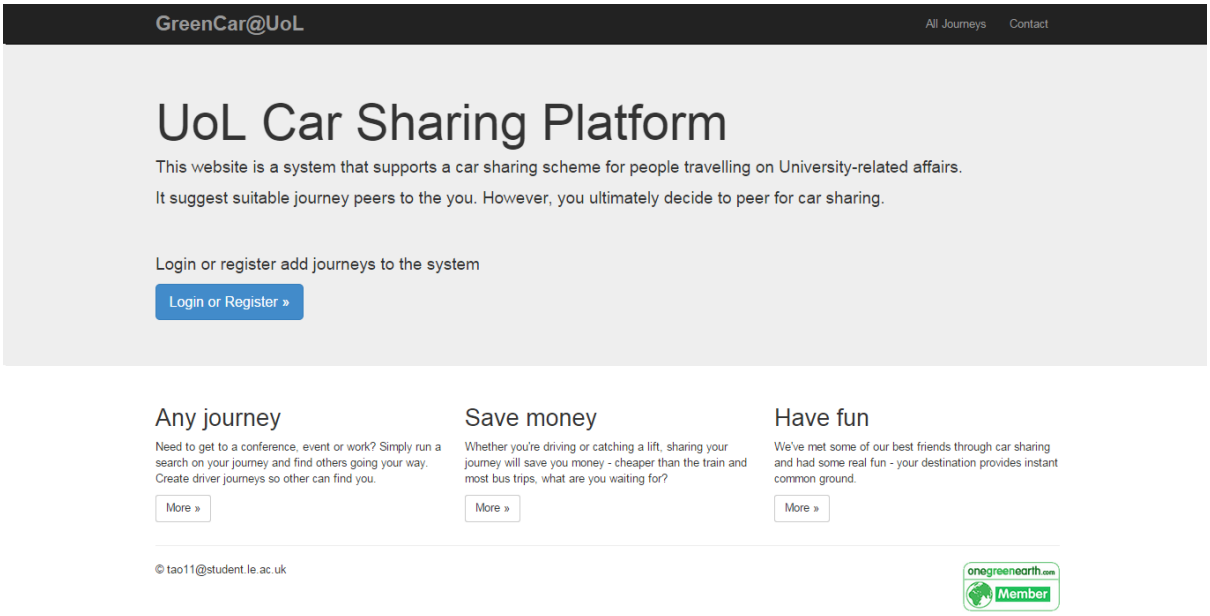

Figure 12: Home page before login

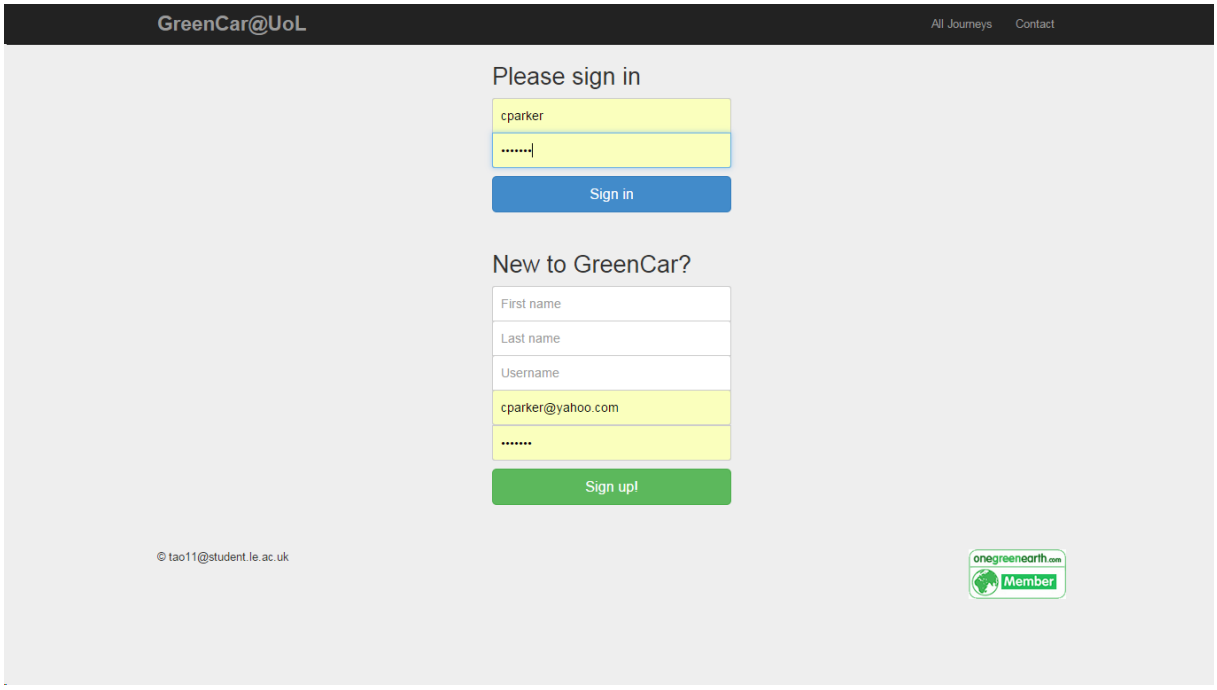

Figure 13: Login and registration page

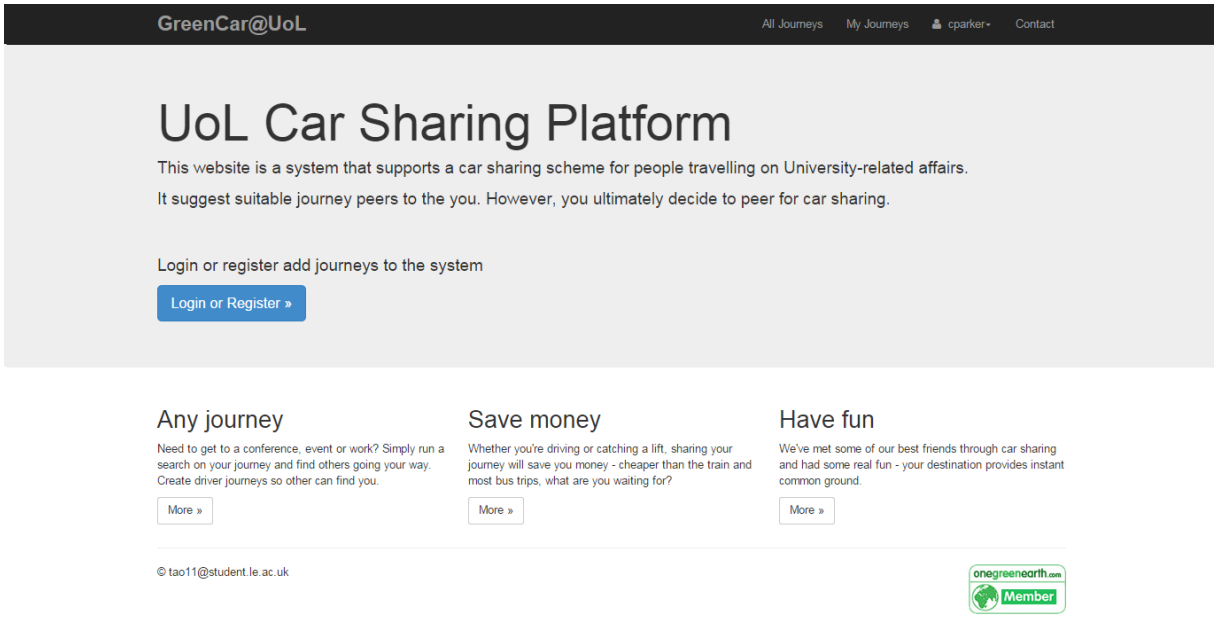

Figure 14: Login page after login

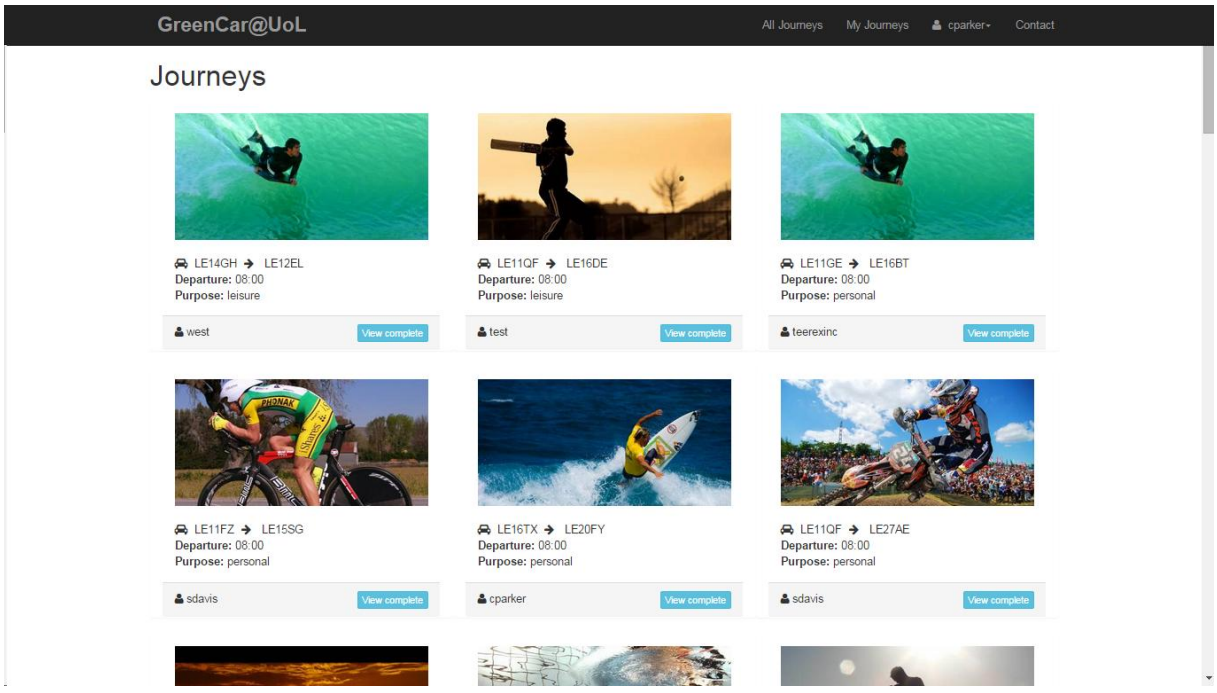

Figure 15: All journeys view

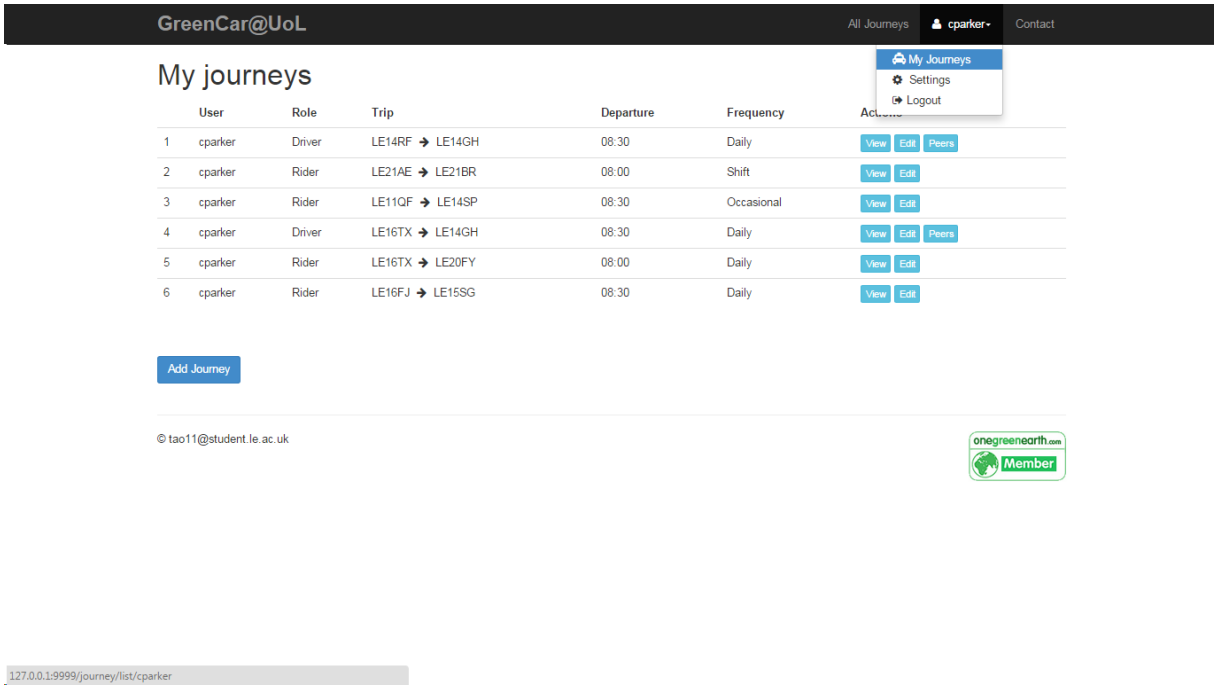

Figure 16: My journeys view

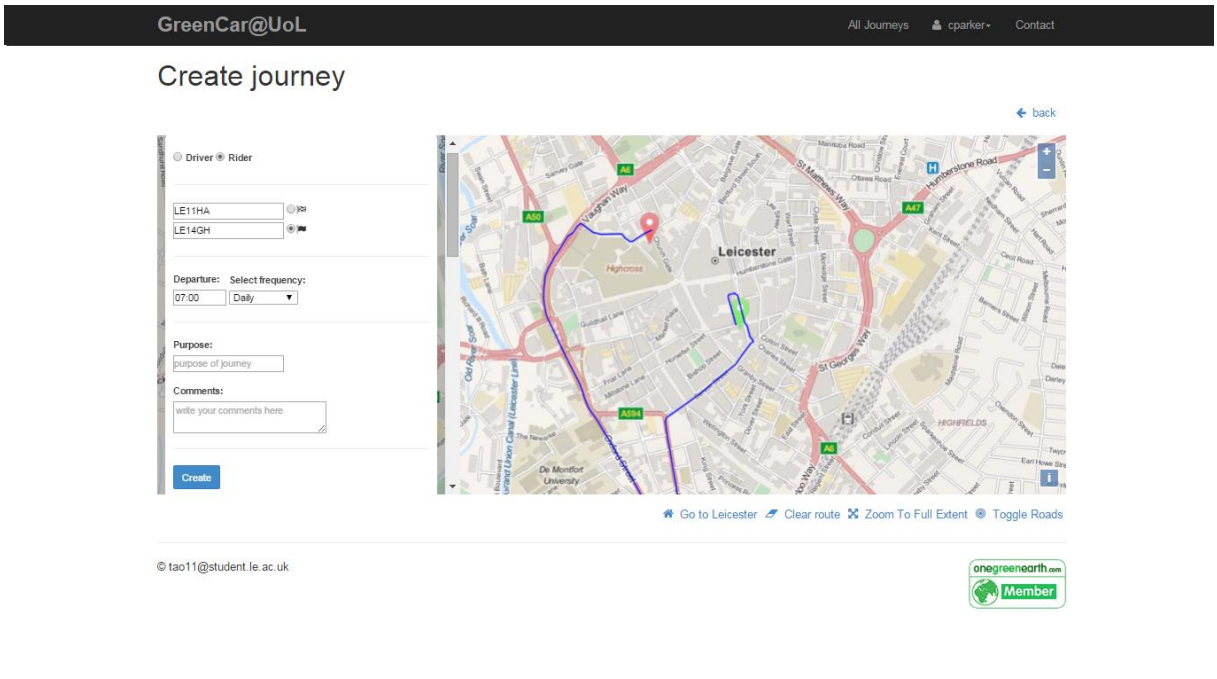

Figure 17: Create journey view

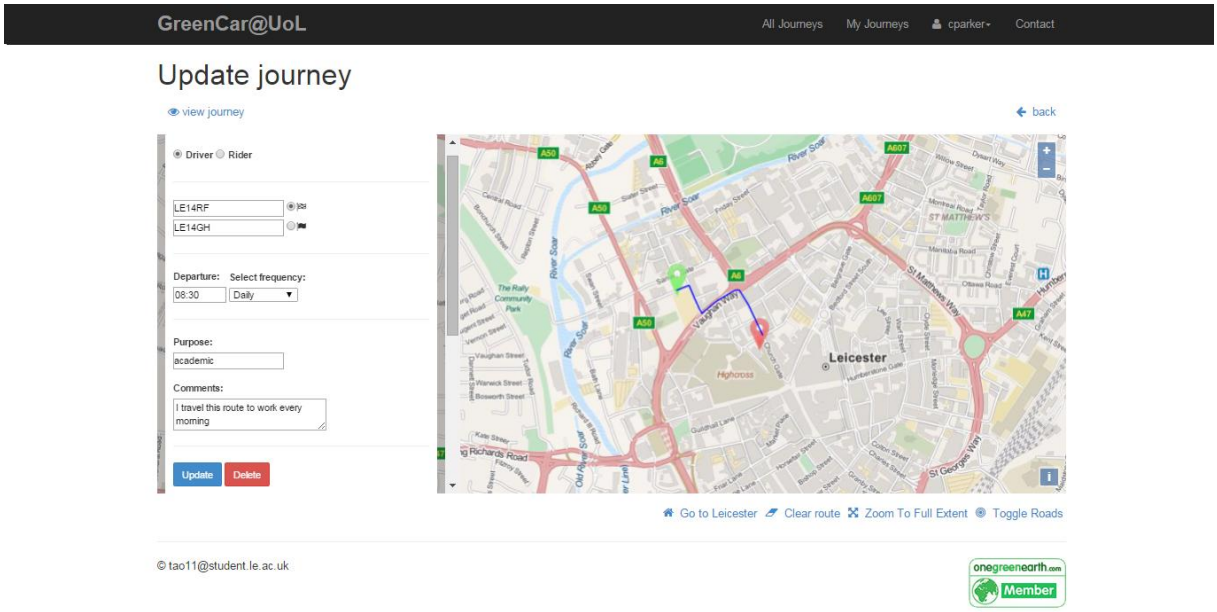

Figure 18: Update journey view

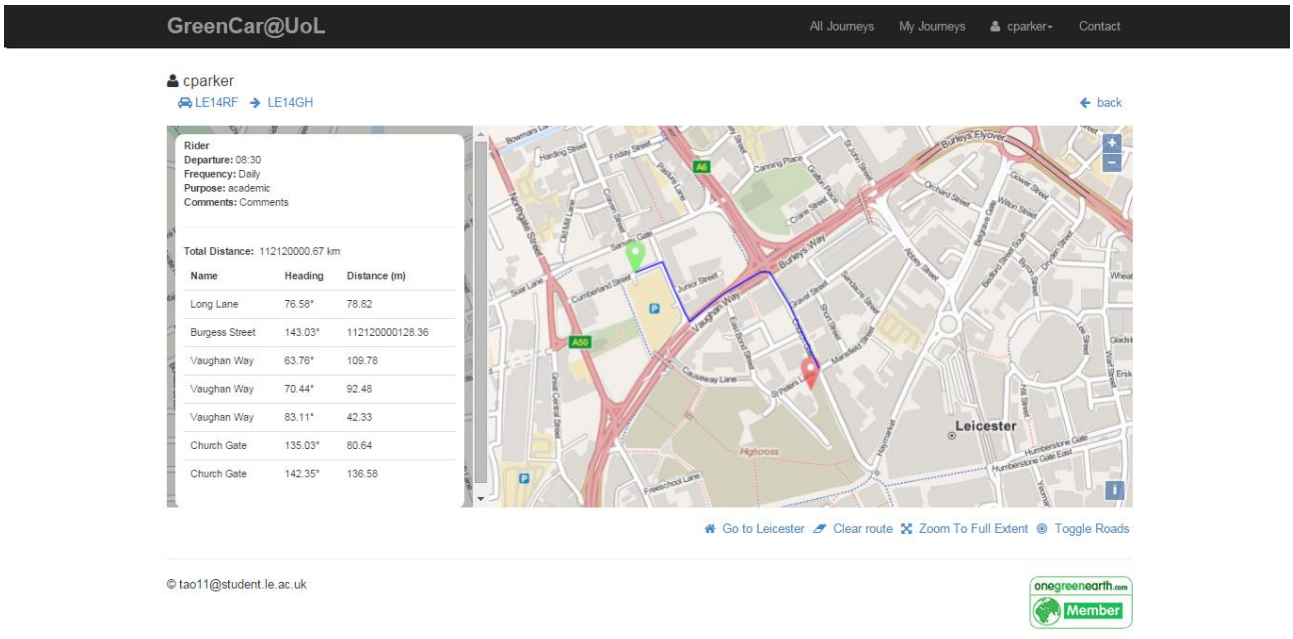

Figure 19: Journey view complete

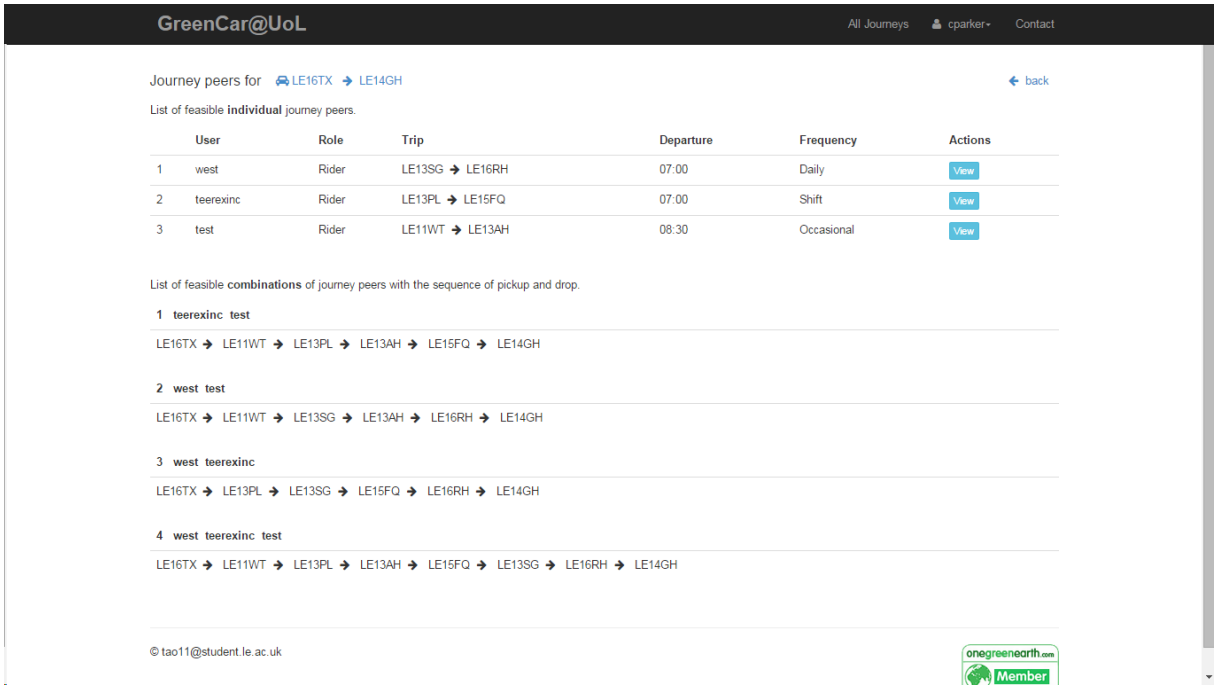

Figure 20: Journey peer view

5.4 Installation instructions

The software is built with the option of cloud deployment. The software can be built from source by carefully following the instructions in this section. The source code is based on a couple of getting started code found on the web (see Appendix 5 for a list of reference boiler plate codes used during implementation).

5.4.1 Setting up locally

1. Clone the project or download from Subversion

```
$ git clone https://bitbucket.org/teerexinc/greencar.git
```

2. Change directory to project

```
$cd greencar
```

3. Download and install PostgreSQL

4. Create a database named "greencar" and set up routing component see Appendix 1.

5. Change dbURL bean in db-psql.xml using the credential of the database created.

```
value="postgres://[USERNAME]:[PASWORD]@[HOST]:[PORT]/[DATABASE_NAME]"
```

6. Download and install Maven

7. Run the app

```
$mvn -D jetty.port=9999 jetty:run
```

8. Open the app http://localhost:9999/ in a web browser

5.4.2 Deploying the build to Heroku

1. Create a Heroku account and download toolbelt
2. Create an instance on Heroku (cloud machine referred to as Dynos)

```
$heroku create
```

3. Setup a free instance of PostgreSQL database server on heroku

```
$heroku addons:create heroku-postgresql:hobby-dev
```

4. Check if database environment variable is created (DATABASE_URL should be set)

```
$heroku config
```

9. Change dbURL bean in db-psql.xml using the environment variable set above.

```
value="#{systemEnvironment['DATABASE_URL']}"
```

5. Commit the code

```
$git add --all

$git commit --all –message="Initial commit"
```

6. Push the code to Heroku

```
$git push heroku master
```

7. Open the app

```
$heroku open
```

5.5 Integration

The MVC architecture used on the project provides seamless integration of the user interface with the application backend. On the other hand, integrating the routing component with the web application and map user interface requires careful setup of technologies to interface between the components.

5.5.1 Map user interface – Backend integration

The Map GUI allows users to select points on the map; points on the map user interface represented as latitude and longitude while a point in the Journey entity is represented as a postcode. The postcode.io JSON API is used to convert GIS points to Journey entity table

points and vice-versa. The component GeoToPostcode.java houses the code base for converting between the data types in the backend (see Appendix 3) while the AJAX calls to the postcode.io are used on the map GUI (see Appendix 4).

5.5.2 Routing component – Map user interface integration

The routing component explained in section 5.2 contains functions that return table rows. The rows need to be converted to a map image that can be projected on a layer in the map GUI. GeoServer WMS is the integration tool used to interface between the routing component and the map GUI. Any layer setup in GeoServer is available over WMS; the routing layer is setup as an SQL view layer with the code below. `ST_MakeLine` creates a geometric line from a set of geometric points; `gcar_shortestpath` is a custom SQL function in the routing component, it returns a set of points that make up the shortest path between the points in its argument.

```
1. SELECT ST_MakeLine(route.geom) FROM (SELECT * FROM gcar_shortestpath('roads',
   %x1%, %y1%, %x2%, %y2%) ORDER BY seq) AS route
```

Table 6: SQL staatement for GeoServer view layer

In the map UI, OpenLayers provide a means of sending request to GeoServer WMS and displaying the result on the UI. The route object - an instance of `ol.layer.Image` can be projected on a map.

```
1.     var params = new Object();
2.     params.LAYERS = 'greencar:shortest_path';
3.     params.FORMAT = 'image/png';
4.     params.viewparams = viewparams.join(';');
5.
6.     var route = new ol.layer.Image({
7.         source: new ol.source.ImageWMS({
8.             url: 'http://[host_name]/geoserver/greencar/wms',
9.             params: params,
10.            serverType: 'geoserver'
11.        })
12.    });
```

Table 7: Code extracts from journey.js (integration)

5.6 Testing

This section contains the different test performed to verify the behaviour of the system. Database CRUD operations were tested using JUnit test cases while the web component functionalities were tested for compliance to an expected post-condition given a pre-condition. The SQL functions developed for the routing component was also tested for correctness by comparing the output given a specific input with an expected output.

5.6.1 JUnit test cases

This section describes how to run the JUnit tests and the results of the test.

To set up the test environment locally:

1. Clone the project or download from Subversion

```
$git clone https://bitbucket.org/teerexinc/greencar.git
```

2. Change directory to project

```
$cd greencar
```

3. Download and install Maven

4. Run tests

```
$mvn clean install test
```

Test case	Test description	Result
`saveAJourney()` Tests create journey operation	Create and persist a Journey object j; check if j.id is not null	Pass
`getAJourney()` Test read journey operation	Create and persist a Journey object j; fetch the journey from the database into j_, check if j = j_.	Pass
`getAJourneyUser()` Tests journey-user relationship	Create a Journey object journey, Create and persist a User object user, add user to journey; persist journey; fetch the journey and user from the database; check if journey.user = user	Pass
`listJourneys()` Tests fetching a list of journeys	Create n Journey objects; persist them; fetch all journeys from the database; check if journeys.size = n.	Pass
`listUserJourneys()` Tests fetching journeys by user	Create a User object; create two Journey objects journey and journey_, add user to journey; fetch journeys by user.id; check if journeys.size = 1 and journeys[0].user = user	Pass
`listJourneysByFrequency()` Tests fetching journeys by frequency	Create two Journey objects j and j_, set j.frequency = daily; set j_.frequency = occasional. Persist j and j_; fetch journeys where frequency is daily; check if journeys.size = 1 and journeys[0].frequency = daily.	Pass
`listRiderJourneys()`	Tests fetching rider journeys for a given driver journey.	Pass
`listRiderJourneys_()`	Tests fetching rider journeys using journeys created by same user.	Pass
`deleteAJourney()`	Create and persist a Journey object j; delete the journey; fetch the deleted journey with id from the database and check if it is null.	Pass

Table 8: JUnit tests and results

5.6.2 Requirements test

This section contains results of the test performed on the product. The table shows the software's compliance to relevant requirements by checking if the system results in an expected post-condition given a pre-condition.

Requirement	Pre-condition	Post-condition	Results

System supports multiple users via registered individual User accounts.	None	Two users with different accounts	Pass
A user should be able to log into the system using a registered User account.	A registered user account	Login successful and alert message	Pass
An authenticated user can create many journeys.	A registered user is logged in	User has two or more journey on MyJourneys page	Pass
An authenticated user can update and delete journeys they created	A registered user is logged in; with one journey on MyJourneys page	Journey is updated or deleted	Pass
Users cannot update or delete journeys created by other users.	A registered user is logged in; system has a journey created by another user	Error page with relevant error message	Pass
An authenticated user can view journey peers	A registered user is logged in; user has created one journey with driver modality	List of individual journey peers and list of feasible combinations with driving instructions.	Pass

Table 9: Pre and Post condition of functional requirements

5.6.3 Implementation status

The table below shows the implementation status of relevant non functional requirements.

Requirement	Implementation Status	Related codes
Any Users can view all journeys	Complete	JourneyController.java
An authenticated user can view journey peers for journeys she created with driver modality.	Complete	JourneyController.java; peer/list.jsp; peer/single.jsp
Journey peer matched.	Complete	JourneyPeer.java,
An authenticated user can view journey peers.	Complete	JourneyController.java
Journeys and user accounts detail are stored to and retrieved from a web-based service that is deployable to the cloud.	Complete	Project wide
The system must include a genuine, flexible, cost-efficient mechanism to compute routes for journeys that is trustworthy. Data from OpenStreetMap should be used to compute journey routes.	Complete	journey.js; gcar_shortestpath.sql
The web application client UI should consist of views to manage user accounts (including user preferences), journey	Partial	UserController.java; user/edit.jsp

requests and statistics.		
Performance and scalability	Complete	Project wide
Able to deal with the whole planet map from OpenStreetMap.	Complete	Routing component
Access control must be clearly defined to demarcate operations available to only authenticated users.	Complete	Pass
Web component of the system should expose a RESTful JSON API for operations available to users.	Complete	Pass
Email notification system	Not implemented	-

Table 10: Implementation status of non-functional requirements

6 Conclusion

6.1 Analysis of results achieved during the project

In summary, the projects formalizes the computational problem of finding feasible paths for combinations of journey peers and provides a simple yet efficient algorithm for solving the problem which improves cost savings on single occupancy drives or single journey peer. This research gives a new research perspective to car sharing in the context of computer sciences.

The algorithm is blazingly fast compared to an exhaustive search and its solutions always satisfy the vertex ordering constraints. However, the solution found by the algorithm for a combination of riders is not always the optimal solution for that combination of riders. Overall, the solutions found are very useful and provides cost saving for both drivers and riders compared to the cost of single occupancy drives or car sharing with individual journeys peers.
The project would have benefited more if it was solely theoretical – facilitating more research into advanced algorithm designs for solving the problem. Nonetheless; the software product is outstanding in my opinion (full test coverage, modular and well documented). I suggest the code base is used on any other projects involving GreenCar@UoL.

6.2 Outline of theories learnt during project

1. While integrating the components there was need to use the postcode.io API in JourneyController.java. I learnt how to develop a component that provides a standard interface to with a JSON API with standard result types.
2. I learnt how to develop an enterprise application using Spring Web MVC.
3. While researching how to compute feasible combinations of journey peers, I got deeper understanding of algorithms; complexity theories and problem classes.
4. I learnt how to develop a Java based cloud application.

6.3 Reflections on project

The main challenge of the project was developing a custom library for computing driving directions of journeys. However with careful research, I discovered PGRouting - an open source routing library for computing driving directions at the database layer. In my opinion no significant improvement can be made to what is offered by the library as it supports both A* and Dijkstra's algorithm. Hence, developing a custom routing module on this project would not contribute to the field of computing. Furthermore, all operations that rely on geospatial data was performed on the database layer providing improved performance and modularity.

Computing a feasible path for a combination of journey peers in a car sharing context has not been explored directly in a computer science research before this project. This project formulates the problem of computing a feasible path for a combination of journey peers mathematically as a combination of optimization and precedence constraint satisfaction problems. It also explores a heuristic greedy algorithm for solving the problem given a feasibility upper bound.

The journey peer combination problem is a challenging computational problem and requires more time than was anticipated at the early stages of the project. Devising a suitable algorithm for the problem qualifies as an individual project without additional aims or challenges. Hence, I deleted implementing a communication model based on WebSocket

from the non-functional requirements of the project to facilitate enough time for a thorough research on providing a reasonable algorithm to solve the combination problem.

The changes made to the requirement did not change the project delivery timeline. All implementation plans for WebSocket were easily swapped with HTTP, which is why I chose to delete WebSocket from the non-functional requirements.

7 Bibliography

(1) A. Boronat, GreenCar@UoL Project Proposal University of Leicester: , 2014.

(2) Cranfield University, 2013-last update, Fact and Figures: Environment Available: http://www.cranfield.ac.uk/about/cranfield/facts-and-figures/environment.html [02/26, 2015].

(3) K.E. Foote and M. Lynch, 2014-last update, **Geographic Information Systems as an Integrating Technology: Context, Concepts, and Definitions** Available: http://www.colorado.edu/geography/gcraft/notes/intro/intro.html [02/27, 2015].

(4) M. Garey and D. Johnson, Computers and Intractibility: A Guide to the Theory of NP-Completeness. United States: W. H. Freeman and Company, 1979.

(5) R. Heckel and M. Lohmann, LNCS, 2621, 170, 2003.

(6) I. Heywood, S. Cornelius and S. Carver, An introduction to Geographical Information Systems. fourth edn. Pearson, 2011.

(7) Z. Kiraly and P. Kovacs, Acta Universitatis Sapientiae, Informatica, 4, 67, 2012.

(8) Liftshare.com, 2015-last update, UK's largest car-share/carpooling service website Available: liftshare.com [02/23, 2015].

(9) Open Geospatial Consortium Inc., OpenGIS® Web Map Server Implementation Specification, 1.3.0, Final, 2006.

(10) J. Park, H. Hwang, J. Yun and I. Moon, Advanced Science and Technology Letters, 46, 307, 2014.

(11) D. Pisinger and S. Ropke, Computers & Operations Research, 34, 2403, 2007.

(12) A. Pouly, Graphs: Minimum cost flows France: , 2010.

(13) S. Ropke, Heuristic and exact algorithms for vehicle routing problems, University of Copenhagen, 2005.

(14) A. Schrijver, A Course in Combinatorial Optimization. Amsterdam, Netherlands: , 2013.

(15) R. Sedgewick and K. Wayne, 2014-last update, **Algorithms, 4th Edition** Available: http://algs4.cs.princeton.edu/home/ [21/04, 2014].

(16) S. Shaheen, D. Sperling and C. Wagner, Transportation Quarterly, 52, 35, 1998.

(17) Taylor Lightfoot Transport Consultants, Pay As You Drive Carsharing. EU SAVE CONTRACT NO: 4.1031/Z/95-025. UK: 1995.

(18) M. Thorup, Journal of the ACM, 46, 362, 1999.

(19) W3C Recommendation, 2014-last update, **HTML5: A vocabulary and associated APIs for HTML and XHTML** Available: http://www.w3.org/TR/html5/ [03/11, 2015].

(20) J. White, K. Czarnecki, D.C. Schmidt and G. Lenz, IEEE International Enterprise Distributed Object Computing Conference, 11, 2007.

(21) Wiki.openstreetmap.org, 2015-last update, Planet.Osm - Openstreetmap Wiki Available: http://wiki.openstreetmap.org/wiki/Planet.osm [02/26, 2015].

(22) F. Xie, Y. Jia and R. Jia, Journal of Convergence Information Technology, 7, 21, 2012.

Appendix

1. Routing module setup

```
-- At this point, it is assumed that user (administrator) has a working installation of PostgreSQL database
add extension postgis;

-- import shapefiles for roads using PostGIS shapefiles loader it is pertinent to use the correct SRID - 4326

-- follow the instructions at http://workshop.pgrouting.org/index.html to install PGRouting

-- create topology of OSM road data in the roads table
SELECT pgr_createTopology('roads', 0.00001, 'geom', 'gid');
-- The result should contain
"NOTICE:Vertices table for table public.roads is public.roads_vertices_pgr”

-- Prepare roads  table for routing
ALTER TABLE roads ADD COLUMN "source" integer;
ALTER TABLE roads ADD COLUMN "target" integer;
CREATE INDEX roads_source_idx ON roads("source");
CREATE INDEX roads_target_idx ON roads("target");

-- Add a column for keeping the cost of one-way streets
ALTER TABLE roads ADD COLUMN "length" double precision;
UPDATE roads SET length = ST_Length(geom);
ALTER TABLE roads ADD COLUMN "reverse_length" double precision;
UPDATE roads SET reverse_length = length WHERE oneway = 0;
- Make the length of one-way streets large
UPDATE roads SET reverse_length = 999 WHERE oneway = 999;
```

2. Security and access control (extract from security-context.xml)

```
<http pattern="/resources/**" security="none"/>
    <http pattern="/login" security="none"/>

    <http auto-config='true'>
    <intercept-url pattern="/journey/view/**" access="ROLE_ANONYMOUS"/>
        <intercept-url pattern="/journey/**" access="ROLE_USER"/>
        <intercept-url pattern="/user/edit/**" access="ROLE_USER"/>
        <intercept-url pattern="/user/list/**" access="ROLE_ADMIN"/>

        <logout/>

        <form-login login-page='/login'
```

```
                    login-processing-url="/process-login"
                    username-parameter="username"
                    password-parameter="password"
                    default-target-url="/"
                    authentication-success-handler-ref="authenticationSuccessHandler"
                    authentication-failure-handler-ref="authenticationFailureHandler"/>
        </http>
```

3. Code extracts from GeoToPostcodes.java (Backend integration)

```
/**
 * Simple class for converting postcode to geolocation vice-versa based on
 * <a href="http://postcodes.io/">Postcode & Geolocation API for the UK</a>.
 *
 */
  ... some codes missing here ...

    /**
     * Represents the fields in the JSON data for the Postcodes API.
     */
    public static class Result {
        @Key
        public String incode;

        @Key
        public String outcode;

        @Key
        public String postcode;

        @Key
        public Double longitude;

        @Key
        public Double latitude;
    }

    /**
     * Represents a typical result for postcode to geolocation.
     */
    public static class PostcodeToGeoResult extends PostcodesApiBaseResult {
        @Key
        public Result result;
    }

    /**
     * Represents a typical result for geolocation to postcode.
     */
    public static class GeoToPostcodeResult extends PostcodesApiBaseResult {
        @Key
        public List<Result> result;
    }

    public static Point postcodeToGeo(String postcode) throws Exception {
        HttpRequestFactory requestFactory =
                HTTP_TRANSPORT.createRequestFactory(new HttpRequestInitializer() {
                    @Override
                    public void initialize(HttpRequest request) {
                        request.setParser(new JsonObjectParser(JSON_FACTORY));
                    }
                });
        PostcodesApiUrl url = new PostcodesApiUrl(PostcodesApiUrl.API_BASE_URL + "/" + postcode);

        HttpRequest request = requestFactory.buildGetRequest(url);
        PostcodeToGeoResult apiResult = request.execute().parseAs(PostcodeToGeoResult.class);

        if (apiResult.status == HttpStatus.SC_OK) {
```

```
            return new Point(apiResult.result.longitude, apiResult.result.latitude);
        }
        if (apiResult.status == HttpStatus.SC_NOT_FOUND) {
            throw new org.apache.http.client.HttpResponseException(HttpStatus.SC_NOT_FOUND, apiResult.error);
        } else
            throw new org.apache.http.client.HttpResponseException(HttpStatus.SC_INTERNAL_SERVER_ERROR, "Contact developer!");
    }

    public static Result geoToPostcode(Point point) throws Exception {
        HttpRequestFactory requestFactory =
                HTTP_TRANSPORT.createRequestFactory(new HttpRequestInitializer() {
                    @Override
                    public void initialize(HttpRequest request) {
                        request.setParser(new JsonObjectParser(JSON_FACTORY));
                    }
                });
        PostcodesApiUrl url = new PostcodesApiUrl(PostcodesApiUrl.API_BASE_URL + "?lon=" + point.getLon() + "&lat=" + point.getLat());
        HttpRequest request = requestFactory.buildGetRequest(url);
        GeoToPostcodeResult apiResult = request.execute().parseAs(GeoToPostcodeResult.class);

        if (apiResult.status == HttpStatus.SC_OK) {
            return apiResult.result.get(0);
        }
        if (apiResult.status == HttpStatus.SC_NOT_FOUND) {
            throw new org.apache.http.client.HttpResponseException(HttpStatus.SC_NOT_FOUND, apiResult.error);
        } else
            throw new org.apache.http.client.HttpResponseException(HttpStatus.SC_INTERNAL_SERVER_ERROR, "Contact developer!");
    }
}
```

4. Code extracts from journey.js –Map GUI integration

```
// jQuery getJSON is used to make AJAX call to the API
$.getJSON('http://api.postcodes.io/postcodes?lon=' + lon + '&lat=' + lat, function (data) {
      if (null !== data.result) {
          source.setGeometry(new ol.geom.Point(event.coordinate));
          var result = data.result[0];// get first results
          var postcode = result.outcode + result.incode;
      }
});
```

5. Code base of this project is based on the following reference boiler plate codes

- https://github.com/ParallelBrains/Spring-Hibernate-PostgreSQL-Maven-Jetty
- https://github.com/nicolatassini/Heroku-J2EE-Spring-MVC---Hibernate---PostgreSQL----Maven---Jetty
- https://github.com/heroku/devcenter-spring-mvc-hibernate
- https://devcenter.heroku.com/articles/getting-started-with-spring-mvc-hibernate
- http://algs4.cs.princeton.edu/40graphs/
- http://workshop.pgrouting.org/index.html