# Android App for Ferry Route Optimization

Name: Stephen Jacob

ID: 11121556

Supervisor: Dr. Norah Power

# Abstract

### Existing System

The company Derry Bros Shipping Ltd handles reservations for deliveries by lorry to or from Ireland that require one or two ferry crossings, one for the UK, possibly two for continental Europe. The company plans the route for the driver. In lots of cases, this is sufficient and gets the lorry driver to and from their destination.

Sometimes routes supplied by Derry Bros have to be adjusted due to a number of reasons. For example, a ferry crossing might be cancelled due to bad weather. Traffic conditions might cause a lorry to be late for a scheduled ferry crossing. What normally happens is that the lorry driver would ring the Derry Bros company and request an alternative route.

### Stakeholders

The stakeholders are:

* Derry Bros Shipping Ltd
* Derry Bros Shipping Ltd clients which are lorry drivers.

### The Problem

The problem is

* When ferries are cancelled this can result in Derry Bros being swamped with request calls from drivers for alternate crossings to their destination.
* Requests for adjusted routes at night can require Derry Bros have staff available to consult with ferry crossing information and find an available route.

### Solutions

The first solution is for the company to hire enough staff to deal with these calls. Also it needs staff available to be on call at night. This is an expensive solution and the company want to find a more cost effective approach.

The alternative is to make the information available to lorry drivers. This isn’t enough on its own as the driver or driver’s company are paying Derry Bros for a service, not just for information. What is needed is an application that can access ferry crossing information and travel times on roads and find the optimal route.

The result of such an application would be:

* Better service to the lorry drivers as best route can be found in a couple of seconds rather than having to waiting to get through on a phone call to Derry Bros when lots of other lorry drivers are trying to get through as well
* The Derry Bros company can get by with less staff because of the fact that they’re not going to be swamped with phone calls for alternative routes. Also existing won’t be worked off their feet when drivers make such calls.

### The Application

The solution would be that the lorry drivers would have a mobile application installed on their phone and they will use this to determine the optimal route given the driver’s current location and intended destination.

This application gets information from two REST sources: the crossing data and the Google Maps APIs. Both sources will be accessed using Http GET Requests.

The application will take the user’s preferences for the optimal route into account, i.e. less driving, shorter ferry crossing or less overall travelling time.

Contents

[Android App for Ferry Route Optimization 1](#_Toc415838511)

[Abstract 2](#_Toc415838512)

[Existing System 2](#_Toc415838513)

[Stakeholders 2](#_Toc415838514)

[The Problem 2](#_Toc415838515)

[Solutions 2](#_Toc415838516)

[The Application 3](#_Toc415838517)

[Introduction 7](#_Toc415838518)

[1.1 Background 7](#_Toc415838519)

[1.2 The Problem 7](#_Toc415838520)

[1.3 Aims/Objectives 7](#_Toc415838521)

[2. Problem Frames 9](#_Toc415838522)

[2.1. The Travelling Salesman Problem 9](#_Toc415838523)

[2.2 Vehicle Routing Problem 9](#_Toc415838524)

[2.3 Decision Trees 10](#_Toc415838525)

[2.4 Visitor Pattern 10](#_Toc415838526)

[3. Application Prototype 13](#_Toc415838527)

[3.1 Requirements 13](#_Toc415838528)

[3.1.1 Functional Requirements 13](#_Toc415838529)

[3.1.2 Non-Functional Requirements 13](#_Toc415838530)

[3.2 Prototyping 13](#_Toc415838531)

[3.2.1 First Prototype – Model Classes & Decision Tree algorithm 13](#_Toc415838532)

[3.2.2 Second Prototype – Local Postgres Database 14](#_Toc415838533)

[3.2.3 Third Prototype – Heroku REST Application 14](#_Toc415838534)

[3.2.4 Fourth Prototype – Java Client parses the XML 14](#_Toc415838535)

[3.2.5 Fifth Prototype – Port Java Client to Android 14](#_Toc415838536)

[3.3 Logical Design 14](#_Toc415838537)

[3.4 Physical Design 15](#_Toc415838538)

[3.4.1 Client Structure 15](#_Toc415838539)

[3.4.2 Server Structure 17](#_Toc415838540)

[3.5 UML Class Diagrams 18](#_Toc415838541)

[3.5.1 Stage 18](#_Toc415838542)

[3.5.2 Place 19](#_Toc415838543)

[3.5.3 Route 19](#_Toc415838544)

[4. Software Components 20](#_Toc415838545)

[4.1 REST Web Services 20](#_Toc415838546)

[4.1.1 JAX-RS 21](#_Toc415838547)

[4.1.2 JAXB 23](#_Toc415838548)

[4.1.3 XMLPullParser 24](#_Toc415838549)

[4.2 Public APIs 25](#_Toc415838550)

[4.2.1 Google Distance Matrix API 25](#_Toc415838551)

[4.2.2 Google Maps API 26](#_Toc415838552)

[4.2.3 Location (GPS Coordinates) from the Mobile Device 27](#_Toc415838553)

[4.3 Cloud Deployment 28](#_Toc415838554)

[4.3.1 Git 29](#_Toc415838555)

[4.3.2 Maven 29](#_Toc415838556)

[4.3.3 Heroku 30](#_Toc415838557)

[4.4 SQL Database 31](#_Toc415838558)

[4.4.1 Postgres 32](#_Toc415838559)

[4.5 org.joda.time 33](#_Toc415838560)

[5. Implementation 35](#_Toc415838561)

[6. Testing 38](#_Toc415838562)

[6.1. Model Classes & Decision Tree algorithm 38](#_Toc415838563)

[6.1.1. Test Cases 38](#_Toc415838564)

[6.2. Local Postgres Database 38](#_Toc415838565)

[6.2.1. Test cases 38](#_Toc415838566)

[6.3. Heroku REST Application 39](#_Toc415838567)

[6.3.1. Testing 39](#_Toc415838568)

[6.3.2. Regression Testing 39](#_Toc415838569)

[6.3.2. Test Cases 39](#_Toc415838570)

[6.4. Java Client parsing the XML 39](#_Toc415838571)

[6.4.1. Test Cases 40](#_Toc415838572)

[6.4.2. Regression Testing 40](#_Toc415838573)

[6.5. Android Client 40](#_Toc415838574)

[7. Conclusion 41](#_Toc415838575)

[8. References 44](#_Toc415838576)

# Introduction

## 1.1 Background

This project originated from my Coop placement, from an idea of my supervisor for my Coop, Dr. Enda Fallon, who works in the Software Research Institute (SRI) at the Athlone Institute of Technology (AIT). Dr. Fallon outlined a problem faced by the company Derry Bros Shipping Ltd. It is likely that at some point in the future, this will result in collaboration between the SRI and Derry Bros. The project has been given to me as an interesting problem for a final year project. There is no formal requirements specification and it is not seen that I would have any formal part to play in a future collaboration. I am simply carrying out exploratory work in the area.

## 1.2 The Problem

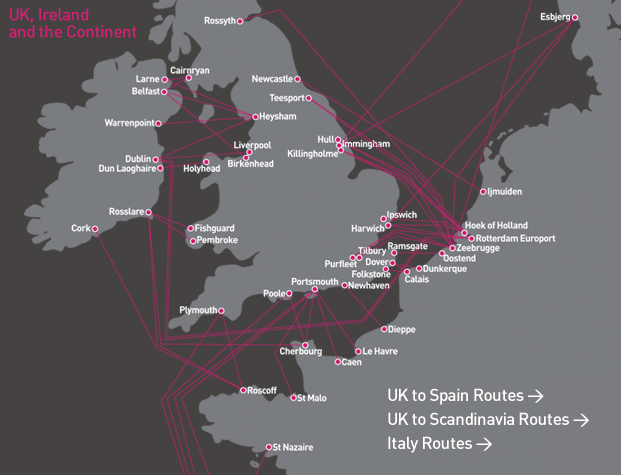
The company Derry Bros handles reservations for (lorry) deliveries from or to Ireland that require one or two ferry crossing, one for the UK, possibly two for continental Europe. The problem addressed by the application is the following. If a ferry crossing is cancelled due to bad weather, this affects lots of lorry drivers, and can result in the company being swamped with request calls from lorry drivers for alternative route plans. Also, lorry drivers could be late for their scheduled ferry due to bad traffic and again require an alternative route plan.

The solution the company would like is that the lorry drivers would have an application installed on their phone and will use this to determine an alternative route.

## ****1.3 Aims/Objectives****

**The aim of the project is to develop an Android application to provide alternatives and make recommendations to lorry drivers in the case where ferry crossings are cancelled or if a driver were to miss the departure time of their ferry due to, for example, bad traffic. The problem to be solved is an optimization problem to optimize various costs such as time or money for the route to the destination.**

**The following is a map of ferry routes taken from the company website.**



# 2. Problem Frames

The goal of the project is to optimize ferry routing for truck drivers. To prepare for the optimization part of the problem generic or reusable problem frames were examined. For instance, the travelling salesman problem and the vehicle routing problem are solutions to a problem where an exhaustive search is not possible for optimal results. Sections 2.1 and 2.2 respectively cover these two solutions, while a solution that implements an exhaustive search is covered in Section 2.1.3. Lastly section 2.4 details a reusable object-oriented pattern that is applicable with the workflow of the project.

## 2.1. The Travelling Salesman Problem

One of the topics reviewed is the travelling salesman problem (TSP). The TSP asks the simple question, find the shortest possible route between a set of cities while visiting each city exactly once and returning to the starting point [Mal1].

The TSP is an NP-hard problem in combinatorial optimization. Suppose there are N cities. The number of possible paths visiting each city exactly once is N!. N! grows very quickly and it is not possible to do an exhaustive search of all routes to obtain the shortest path.

## 2.2 Vehicle Routing Problem

Another optimization algorithm that was researched was the Vehicle Routing Problem (VRP). The VRP is concerned with the situation where vehicles must service a number of drop-off and pick-up locations. The VRP is an important factor in the fields of distribution, transportation and logistics [NEO1].

## 2.3 Decision Trees

A problem solving algorithm that will be used for the project is the decision tree. The decision tree is a decision support tool that models tree-like graphs or nodes of decisions and their consequences. An example of a decision tree is shown below:

Limerick -> Larne -> CairnRyan -> London

-> Heysham -> London

-> Dublin -> Heysham -> London

-> Liverpool -> London

This is a solution based on an exhaustive search on all possible routes. It is possible because the total number of routes is restricted due to routes being funnelled through ports. From one location in Ireland to any destination in England like London for example there are only six possible ports in Ireland to take a ferry to and then perhaps a crossing to only two or three ports in England.

## 2.4 Visitor Pattern

The visitor behavioural pattern is an object-oriented software design pattern [Vis1]. This pattern is used to introduce an operation to be applied to certain elements of a software object structure. The pattern allows for the operation to be implemented without changing the structure of the elements it operates on.

While en route, users of the application will be able to select the cost metric preferences for their optimal route whether it is elapsed time, less time on a ferry crossing or less time driving on a road while on the journey.

The visitor pattern is used in the application as follows. The Visitor interface is called CostMetric.

public interface CostMetric {

**int** getRouteCost(Route r);

}

One example of a ConcreteVisitor is ElapsedTimeMetric. Others are LessDrivingMetric and LessSailingMetric. LessDrivingMetric weights time spent driving more than time spent sailing while LessSailingMetric does the opposite obviously.

**public** **class** ElapsedTimeMetric **implements** CostMetric {

@Override

**public** **int** getRouteCost(Route r) {

**int** cost = 0;

**for**(Stage s: r.getStages()){

cost += s.getElapsedTimeMinutes();

}

**return** cost;

}

}

The class Route accepts the visitor which in this case is the metric. A Route holds a list of Stages and makes a callback to the visitor/CostMetric class to calculate the Route’s value. It passes “this” to the getRouteCost() method in the concrete visitor class.

class Route {

// .....

// visitor pattern

**public** CostMetric metric;

**public** **void** accept(CostMetric metric){

**this**.metric = metric;

}

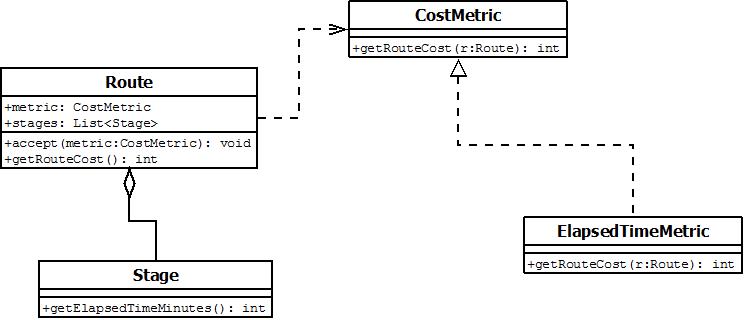
**public** **int** getRouteCost(){

**return** metric.getRouteCost(**this**);

}

}

The route the app has optimized for the driver will have now be taking the driver’s metric preference into account. The UML is shown below.



# 3. Application Prototype

## 3.1 Requirements

Following a meeting with Dr. Enda Fallon in early September a loose set of requirements were agreed for the project. At a subsequent meeting involving a demo of a prototype the requirement for a number of metrics was agreed.

### 3.1.1 Functional Requirements

* The application should supply a driver with a list of possible routes from their current location in Ireland to their intended destination in the UK including one ferry crossing.
* It should sort these routes in ascending order according to the elapsed time taken for the route.
* It should offer the user a number of alternative metrics on which to optimize the route they take.
* It should present the routes in ascending order based on the metric chosen.
* It should present the stages of the route chosen by the user with all departures and arrival times.
* It should display a map overhead of the road stages of the route chosen with the route highlighted.

### 3.1.2 Non-Functional Requirements

* It should be very easy to use as the intended users are lorry drivers with limited IT skills.
* It should return the results in a reasonable amount of time i.e. three to five seconds.

## 3.2 Prototyping

In order to develop the application it was done using a series of prototypes.

### 3.2.1 First Prototype – Model Classes & Decision Tree algorithm

The first prototype was a Java implementation. This included the model classes such as Stage, RoadStage, Crossing, Route, Port, LandLocation and Location. In the main method, instances of these classes were instantiated and linked together. The DecisionTree algorithm was implemented and tested.

### 3.2.2 Second Prototype – Local Postgres Database

The second prototype used a local Postgres database. SQL was used to insert data into the database tables. The information about ports, crossings, road stages was all stored in the database. Data Access Object classes were written for the database including PortDAO and CrossingDAO. The main method then applied the DecisionTree algorithm to the data accessed from the database.

### 3.2.3 Third Prototype – Heroku REST Application

The third prototype moved the data access code (DAOs) into a Heroku REST application which used JAX-RS to route requests to Java methods. The application accepted requests for information about ports and crossings. It read the data from the database into model objects. JAXB annotations serialized the objects into XML.

### 3.2.4 Fourth Prototype – Java Client parses the XML

The client side Java application now sends REST requests to the Heroku application and reads the XML back. It parses the XML and creates new instances of the model objects. The DecisionTree algorithm now works on these objects.

### 3.2.5 Fifth Prototype – Port Java Client to Android

The Java client was ported to Android. Calls are now made both to the Heroku application and the Google Maps API.

## 3.3 Logical Design

The following is an architecture diagram for the final application. The crossing data service and the Android client are implemented as part of the project. The Google APIs are public.

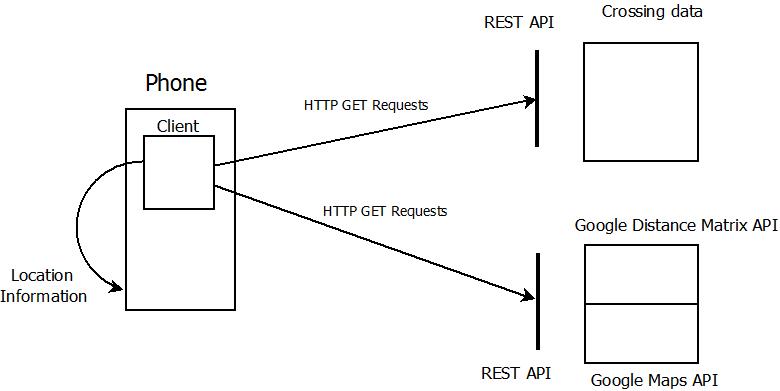


Figure Logical Design

The client gets its current GPS location. The user will type in their intended destination. The application will make GET request to the Heroku application for Crossing data as well as the Google APIs for map information. The client determines the optimal routes using this location. The user is then returned a list of optimal routes to their destination.

## 3.4 Physical Design

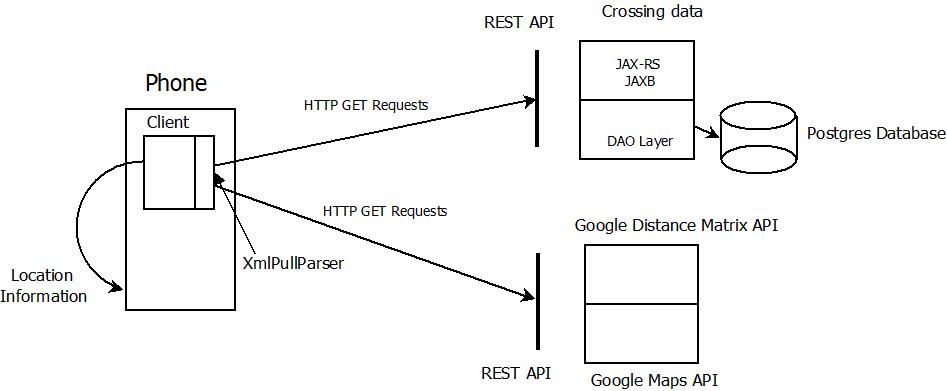


Figure Physical Design

### 3.4.1 Client Structure

The client has been implemented in three layers.

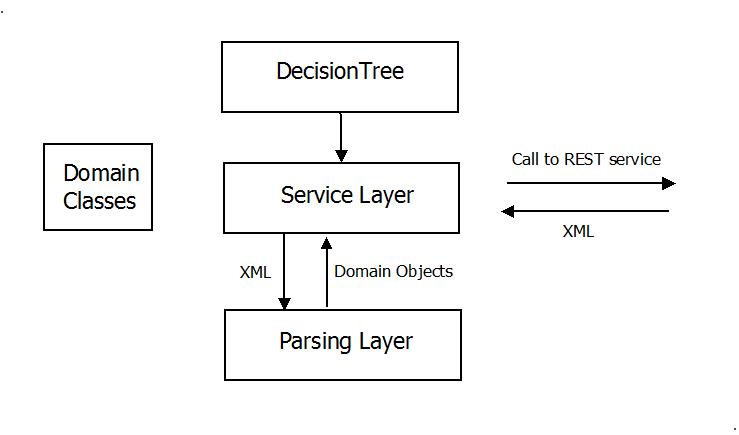


Figure : Client Structure

The top (main) layer is

DecisionTree.buildTree()

The algorithm class DecisionTree.*buildTree()* is the method that defines the decision tree, the decision support tool that optimizes the routes and the stages of each route the lorry driver takes to reach their intended destination.

The second (service) layer is:

DistanceMatrixService.getStages()

PortsCrossingService.getPortsByCountry()

PortsCrossingService.getCrossingsByPort()

At the second layer, the DistanceMatrixService method returns a list of road stages which the destination or departure is a port. The stages could be either all the stages from a single land location to every port in a country, or all the stages from every port to a single landmark.

The PortsCrossingService class serves to return all the ports of a particular country or all the available ferry crossings from a single port.

The third (parsing) layer is;

ParseCrossings.doParseString(String s)

ParsePorts.doParseString(String s)

ParseRoadStages.doParseString(String s)

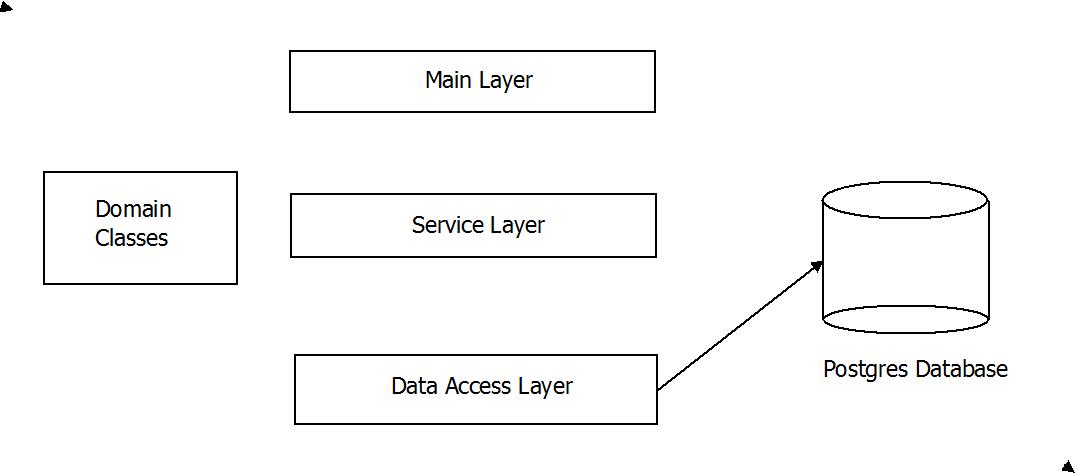
Starting from the third layer, the class ParseCrossings is responsible for parsing the XML returned from a call to the Crossing Service, in particular to get all crossings from a particular port.

The class ParsePorts parses the XML returned from a call to the Ports Service. A call could be to return a list of ports in a country e.g. all the ports in Ireland.

Finally the ParseRoadStages class parses the XML returned from a call to get all road stages to a particular landscape e.g. Leeds, West Yorkshire, England.

### 3.4.2 Server Structure

The server side of the application is implemented in the cloud computing service Heroku.



Figure

The main layer of the server side of the application consists simply of the class that launches the application, as well as the class being the entry point of the system.

The second layer is the service layer consisting of classes which include the JAX-RS annotations. This layer consists of:

CrossingService.getCrossingsByPort

PortService.getPortsByCountry

These classes allow for the application to implement Representational State Transfer (REST) server requests.

The data access layer is the layer used to get data from the database and consists of

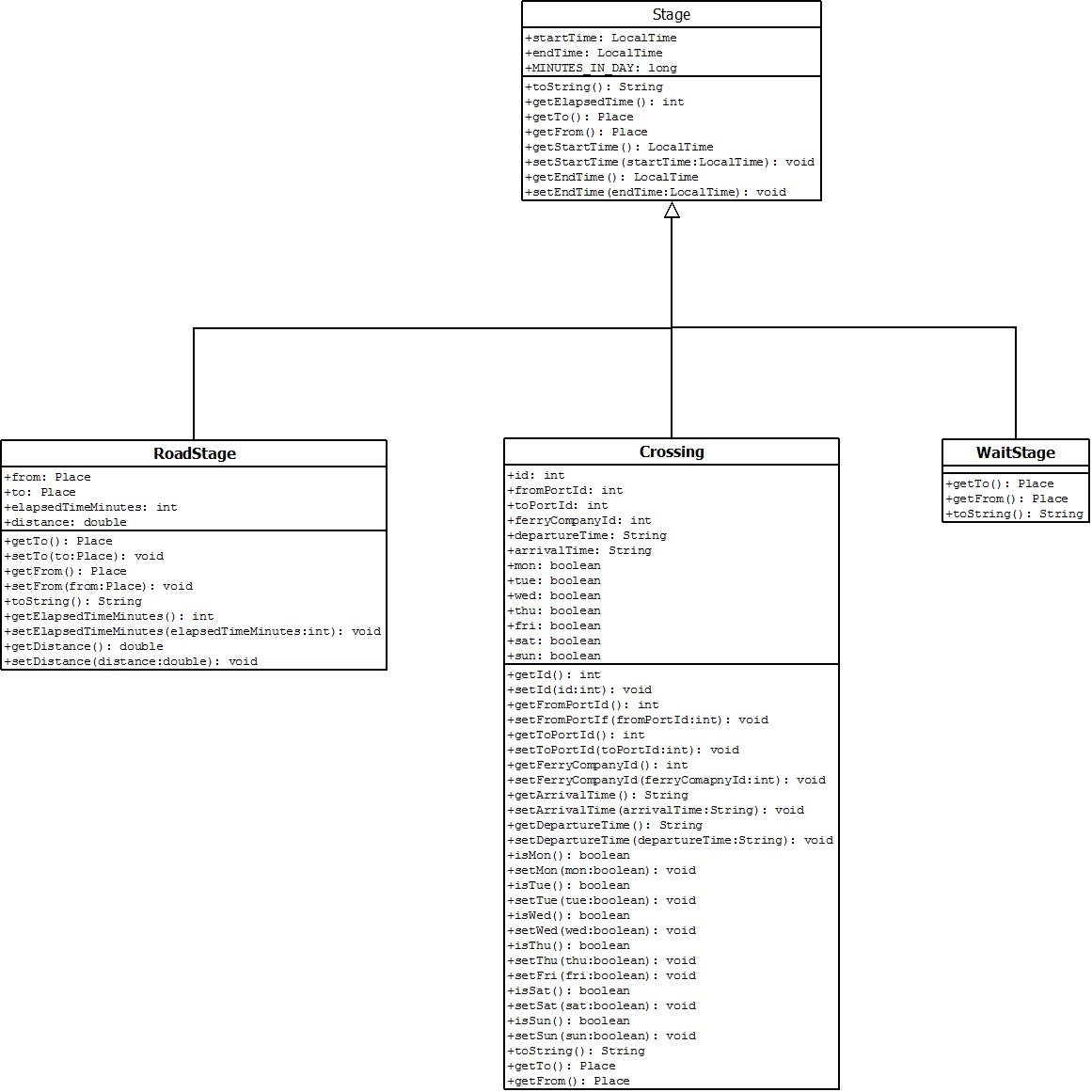
CrossingDAO.listCrossings

PortDAO.listPorts

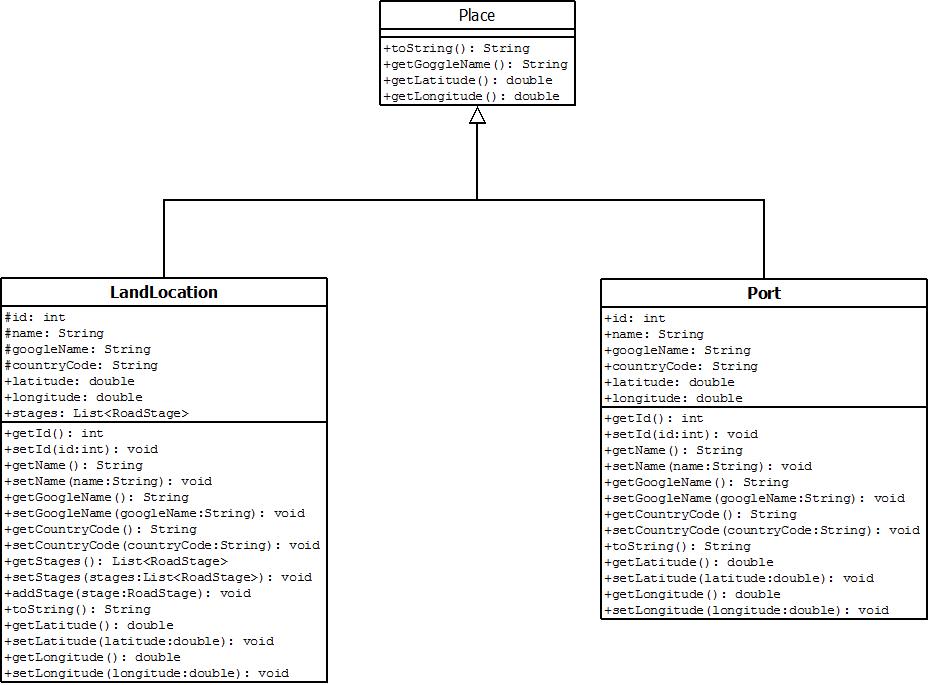
## 3.5 UML Class Diagrams

The following are a series of UML diagrams of the Java classes that comprise the application.

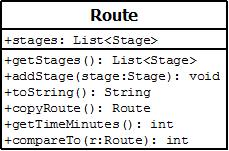
### 3.5.1 Stage



### 3.5.2 Place



### 3.5.3 Route



# 4. Software Components

This section outlines and describes the software used in this project.

Section 4.1 examines the REST architecture, the JAX-RS and JAXB annotations and lastly the XMLPullParser used in Android. The REST architecture will be used to send requests for urls to public APIs that will be used in the project. Section 4.2 looks at these public APIs in particular the Google Distance Matrix and Google Maps API. These APIs can be utilized in a manageable, comprehensive application which can be stored on cloud computing servers. Section 4.3 will detail the use of the Cloud Computing infrastructure in the project. In particular, it looks at Heroku, and also takes a brief look at Git, Maven and public/private keys as these are used for Heroku applications. Section 2.4 looks at the SQL Databases and in particular Postgres as it is the default database for Heroku.

## 4.1 REST Web Services

The following are properties of REST Web services [TIL1]:

* One of the main attributes of REST (Representational State Transfer) is its use of a ‘uniform interface’. For Web Services this meansthe use of HTTP methods for sending requests from clients to servers. These methods include GET, PUT, POST and DELETE. GET is used for reading data, PUT and POST is used for updating data and DELETE is used for deleting resources. In the project, the main one that will be used is GET.
* REST servers provide access to resources. These resources will be commonly identified by URLs.
* Resources will have different representations e.g. XML, JSON. Content negotiation means a client will be able to ask for a specified preferred representation.
* In addition REST servers should be stateless. This means that the server shouldn’t maintain session state after a request has been responded to.
* Finally hypermedia as the engine of application state just means that hyperlinks are used to direct clients to other resources.

The Java specification relevant for REST applications is JAX-RS.

### 4.1.1 JAX-RS

The main issue with writing a REST server is in mapping from or between a URL and a Java method that implements the request. JAX-RS is a Java programming language API that provides support in creating REST web services. This is a two step process as shown below.

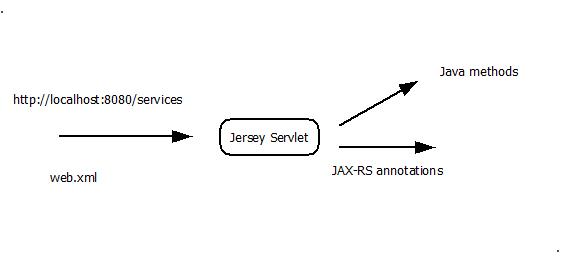


Figure : Routing URLs

The web application is configured so that all URLs of the form <http://servername:port/services> are directed to the JAX-RS implementation class. Using the default Jersey implementation in J2EE this is the class: com.sun.jersey.spi.container.servlet.ServletContainer.

This configuration is done in the web.xml file as shown below:

<servlet>

<servlet-name>jersey-services</servlet-name>

<servlet-class>com.sun.jersey.spi.container.servlet.ServletContainer</servlet-class>

<init-param>

<param-name>com.sun.jersey.config.property.packages</param-name>

<param-value>ie.ul.myproject.services</param-value>

</init-param>

<init-param>

<param-name>com.sun.jersey.api.json.POJOMappingFeature</param-name>

<param-value>true</param-value>

</init-param>

</servlet>

<servlet-mapping>

<servlet-name>jersey-services</servlet-name>

<url-pattern>/services/\*</url-pattern>

</servlet-mapping>

Now annotations are used to map URLs to methods [Vog1]. Consider the following example.

@Path("/port")

**public** **class** PortService {

@GET

@Produces(MediaType.*APPLICATION\_XML*)

@Path("{countryCode}")

**public** List<Port> getPortsByCountry(@PathParam("countryCode") String countryCode) {

**return** PortDAO.*listPorts*(countryCode);

}

}

The annotations used are:

@Path("/port") - all URLs of the form <http://servername:port/services/port/> will be directed to the class PortService.

@GET - the method getPortsByCountry() is called when a GET request is made.

@Produces(MediaType.*APPLICATION\_XML*) - means the output is of type XML.

@Path("{countryCode}") - When a request is made to the URL <http://servername:port/services/port/5>, then the value 5 is matched by the path element {countryCode}.

(@PathParam("countryCode") String countryCode) - the value of the path element countryCode of type String is injected into the method parameter called “countryCode”.

Other annotations exist and are used with PUT, POST and DELETE, but none of those will be used in the project.

### 4.1.2 JAXB

JAXB defines how Java classes are serialized into XML or JSON. Annotations are used [Vog2]. For example:

@XmlRootElement(name = "port")

@XmlType(propOrder = { "id", "name", "googleName", "countryCode" })

public class Port extends Location {

@XmlRootElement(name = "port") – defines the name of the top level element in the XML

@XmlType(propOrder = { "id", "name", "googleName", "countryCode" }) – defines the order in which the attributes are listed in the XML

By default the XML names are the same as the Java attribute names. It is possible to change this default behaviour, using the annotation @XmlElement(name = "newName").

### 4.1.3 XMLPullParser

The Android developer’s website recommends using the XmlPullParser rather than the usual SAX or DOM [And1]. The API is simpler to use than either SAX or DOM[Vog3]. The parser is built into Android but not into J2SE. For prototyping in Java the kxml jar (<http://kxml.org/>) can be used.

The class org.xmlpull.v1.XmlPullParser has a method called next() which returns an int representing the next parsing event. Typical values are

XmlPullParser.*START\_DOCUMENT*

XmlPullParser.*END\_DOCUMENT*

XmlPullParser.*START\_TAG*

XmlPullParser.*END\_TAG*

XmlPullParser.*TEXT*

The method getName() in XmlPullParser then allows to get the name of the tag or gets the text. You set Boolean variables when going into an XML element. This allows one to extract the information from the XML element.

## 4.2 Public APIs

The two public APIs that will be used in the project are Google Distance Matrix and Google Maps.

The Google Distance Matrix allows one to request the distance between a set of origins and a set of destinations and the result is a matrix of the distances and expected times between each origin and destination.

The Google Maps API returns an image overview of an area and is also possible to highlight a route between locations.

### 4.2.1 Google Distance Matrix API

The following is an example of a URL for a Google Distance Matrix request [Gog1].

<http://maps.googleapis.com/maps/api/distancematrix/xml?origins=52.675,-8.571&destinations=53.3461,-6.2083&sensor=false>

The request parameters are:

origins=52.675,-8.571 (University of Limerick)

destinations=53.3461,-6.2083 (Dublin Port)

sensor=false

The above uses GPS coordinates longitude and latitude. It is also possible to specify a location using a name. For example to specify the location for the University of Limerick, the name would be “University+Limerick”.

The response to the above request is:

<?xml version="1.0" encoding="UTF-8"?>

<DistanceMatrixResponse>

<status>OK</status>

<origin\_address>University of Limerick, Unnamed Road, Co. Limerick, Ireland</origin\_address>

<destination\_address>2 Branch Road North, Dublin, Ireland</destination\_address>

<row>

<element>

<status>OK</status>

<duration>

<value>7621</value>

<text>2 hours 7 mins</text>

</duration>

<distance>

<value>210673</value>

<text>211 km</text>

</distance>

</element>

</row>

</DistanceMatrixResponse>

As we can see, this returns a row containing a single element which contains the duration and the distance between the origin and the destination. In the general case there will be multiple rows containing multiple elements. This is a matrix of values.

Another request parameter that can be used is to avoid the use of tolls, in the example below:

<http://maps.googleapis.com/maps/api/distancematrix/xml?origins=52.675,-8.571&destinations=53.3461,-6.2083&sensor=false&avoid=tolls>

This would return the time value of 2 hrs and 19 minutes which is longer than that of when we used the tolls which was 2hrs and 7 minutes. The distance is 197km which is shorter than the earlier example which was 211 km.

### 4.2.2 Google Maps API

This following URL is an example of a request to the Google Maps API.

<https://maps.googleapis.com/maps/api/directions/json?origin=53.4685022,-7.8305376&destination=Dublin+Port&sensor=false>

The request parameters of the previous URL are:

origin=53.4685022,-7.8305376

destination=Dublin+Port

sensor=false

The ‘origin’ parameter represents the GPS coordinates of the location of the application’s current user. The ‘destination’ parameter ‘Dublin+Port’ is a string representation of the user’s intended destination, the local port of Dublin.

This request URL returns a route connecting the two specified coordinates in the JSON software language. The following is a portion of a possible route with the duration between the two coordinates.

"distance" : {

"text" : "133 km",

"value" : 133096

},

"duration" : {

"text" : "1 hour 24 mins",

"value" : 5025

},

"end\_address" : "Dublin Port, Dublin, Ireland",

"end\_location" : {

"lat" : 53.3507568,

"lng" : -6.2145438

},

"start\_address" : "L1437, Co. Westmeath, Ireland",

"start\_location" : {

"lat" : 53.4685954,

"lng" : -7.830433199999999

}

"steps" : [{}, {}, {}, {} ]

A route consists of a number of steps which contains information about each ‘step’ of the journey. This can be used to draw the route on Google Maps. The application UI presents a map overhead of the road stages.

### 4.2.3 Location (GPS Coordinates) from the Mobile Device

The fused location provider is used to return the device’s last known location. To access this, the application uses the permission ACCESS\_FINE\_LOCATION. This is set in the Android manifest file as follows:

<uses-permission android:name="android.permission.ACCESS\_COARSE\_LOCATION"/>

An instance of GoogleApiClient is created using the GoogleApiClient.Builder class as follows.

The Builder adds the current object (this) as a connection callback. The callback method is onConnected(). Inside this method the last known location of the device is obtained as follows.

**public** **void** onConnected(Bundle savedInstanceState) {

System.***out***.println("On connected callback");

android.location.Location *myLastLocation* =

LocationServices.*FusedLocationApi*.getLastLocation(myClient);

Latitude latitude = myLastLocation.getLatitude();

Longitude longitude = myLastLocation.getLongitude();

// .......

}

The latitude and longitude can be obtained from the Location object using getLatitude() and getLongitude()

## 4.3 Cloud Deployment

From the point of view of developers, there are two cloud service options, Infrastructure as a Service (IaaS) and Platform as a Service (PaaS).

IaaS provides physical and virtual machines on which to deploy services. An example of this is Amazons Elastic Compute Cloud (EC2). In order to deploy an application to EC2 would be necessary to install an operating system and solution stack.

PaaS provides a platform or solution stack to which developers can deploy applications. This is suitable for a small application like this project. In particular, what is needed is a platform which consists of a J2EE server and a relational database.

There are a number of such platforms including CloudBees, CloudFoundry and Heroku. The platform chosen in the project is Heroku as for two reasons:

* It is very easy to set up and deploy a REST service using JAX-RS and JAXB.
* It’s possible to create a free account for experimentation and evaluation.

### 4.3.1 Git

Git is a distributed revision control system [Git1]. Git was designed by Linux Torvalds for the Linux kernel. There is a master repository and a local repository. Developers can work independently on their local repository and commit and push/merge changes with master repository.

### 4.3.2 Maven

Maven is a build and dependency management tool used primarily for Java. All Maven projects have a Project Object Model (POM) file. This is an XML representation of the software project, the project lifecycle and a dependency management system for the lifecycle. These would be a superset of features found in other build tools.

For example, from [Mav1]

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<project xmlns=[*http://maven.apache.org/POM/4.0.0*](http://maven.apache.org/POM/4.0.0) xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/maven-v4\_0\_0.xsd">

<modelVersion>4.0.0</modelVersion>

<groupId>org.example</groupId>

<version>1.0-SNAPSHOT</version>

<artifactId>jax-rs-heroku</artifactId>

The POM file defines a groupId, artifactId and version for the project. For example

<groupId>org.example</groupId>

<artifactId>jax-rs-heroku</artifactId>

<version>1.0-SNAPSHOT</version>

It also defines dependencies on other Maven projects. For example,

<dependency>

<groupId>com.sun.jersey</groupId>

<artifactId>jersey-server</artifactId>

<version>1.8</version>

</dependency>

<dependency>

<groupId>postgresql</groupId>

<artifactId>postgresql</artifactId>

<version>9.0-801.jdbc4</version>

</dependency>

These define the dependency on the Jersey servlet (the implementation of JAX-RS) and PostgresSQL which the JDBC Driver for Postgres.

Maven reads the POM file for the project. It downloads required jar files from the central maven repository (<http://mvnrepository.com/>) into the local maven repository ($home/.m2/repository). The download is only done once as the jar files are then found in the local maven repository.

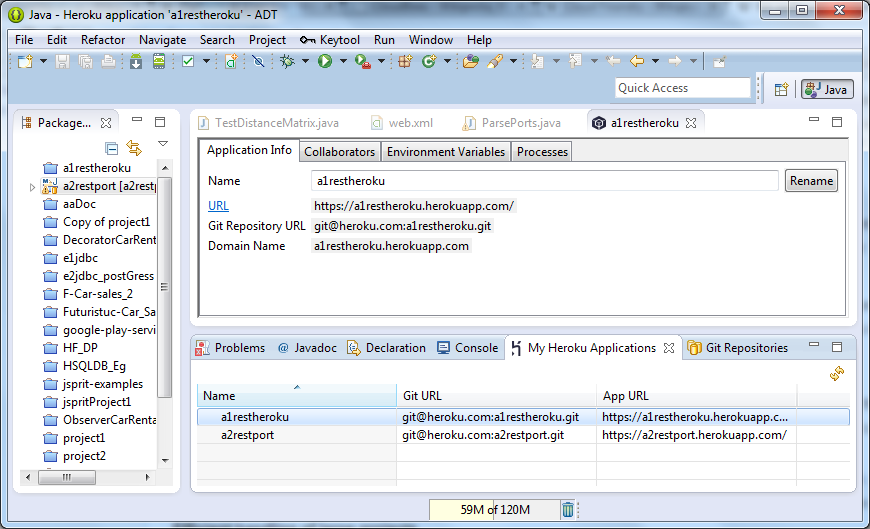
### 4.3.3 Heroku

Heroku has been in development since 2007. It initially supported Ruby but now also supports Java [Her1], PHP, Node.js, Clojure, Python and Scala. The default SQL database that is supported is Postgres [Her3]. It is possible but more difficult to use other databases such as MySQL.

Heroku uses the source code management system Git to deploy applications. It also uses public private key pairs for security checking.

#### Heroku & Eclipse

There is an Eclipse plugin for Heroku [Her2]. The Eclipse update site is <http://eclipseplugin.herokuapp.com/install>. The plugin has an Eclipse view called My Heroku Applications.



This view allows users to see information about Heroku applications, such as name, Git URL and App URL and allows one to deploy, restart, scale and destroy applications and view logs.

#### Heroku & RSA public & private keys

In order to use Heroku it is necessary to have a private (Id\_rsa) and public (Id\_rsa.pub) RSA key and have them put into the directory $home\.ssh.

There is an option in Eclipse to add key to Heroku. The public key is sent to Heroku. In order to deploy and change applications, one must have the corresponding private key.

#### Heroku & GIT Development

First create a REST JAX-RS application. A project is created in the remote Git repository at Heroku [Her4]. A copy of the project also exists in the local Git repository. When we update the project, we must

Commit to the local Git repository

Push to the remote Git repository

These are found under the File-Team-Commit menu. Choose Commit & Push.

#### Heroku & Maven

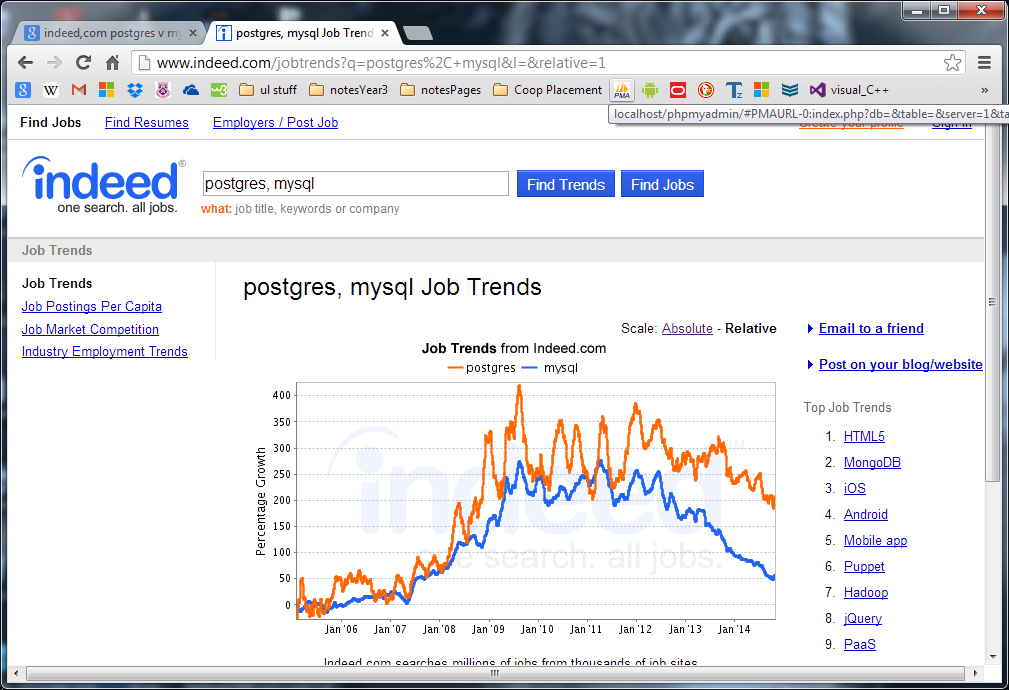
Java projects deployed to Heroku are required to be Maven projects.

## 4.4 SQL Database

For the project, a suitable open-source SQL database would be MySQL. However, the default SQL database for Heroku is Postgres. It is possible to use other databases, but more substantial configuration is required.

Postgres is highly thought of as an open source project but has never had a large corporate sponsor. Also the first function to run on Windows was Version 8. Both these reasons lead to Postgres not being as popular as MySQL. However, data on Indeed.com shows that its rate of growth is currently higher than that of MySQL, but of course from a much smaller base.

Postgres is actually an object-relational database management system. In this we only use the relational features.



In summary, the main reason we’ve chosen Postgres is because it’s the default database with Heroku.

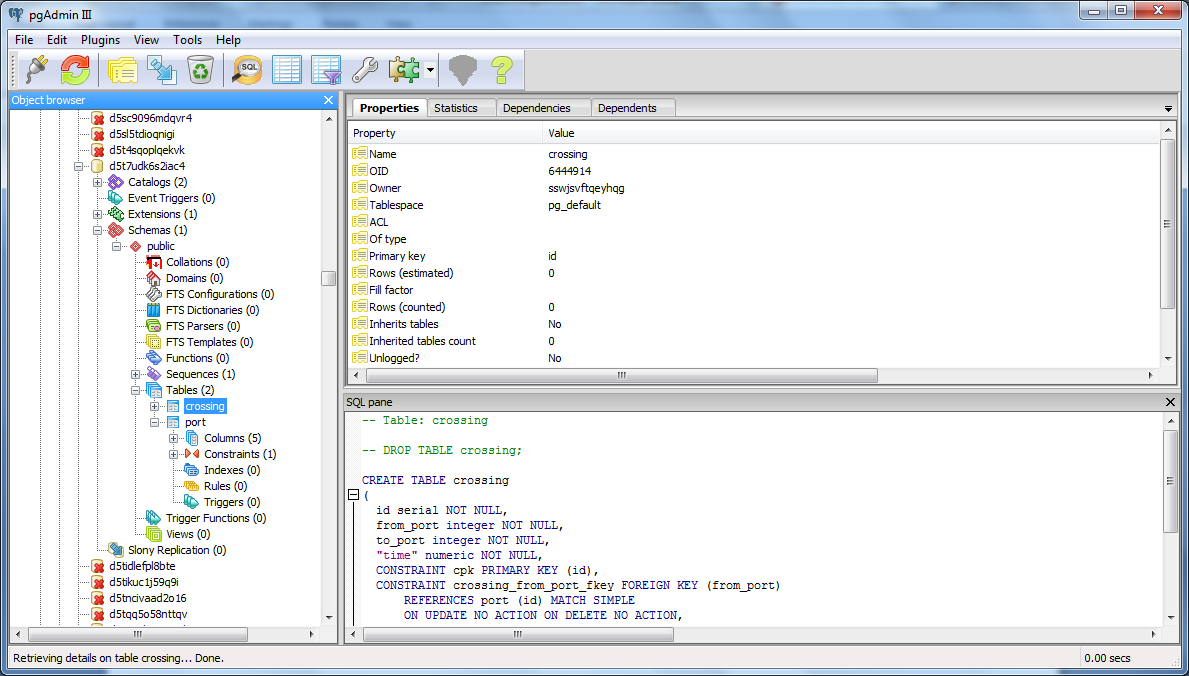
### 4.4.1 Postgres

Postgres uses a GUI-based admin tool for administration called pgAdmin Tool. It is similar in functionality to MySQL Admin.

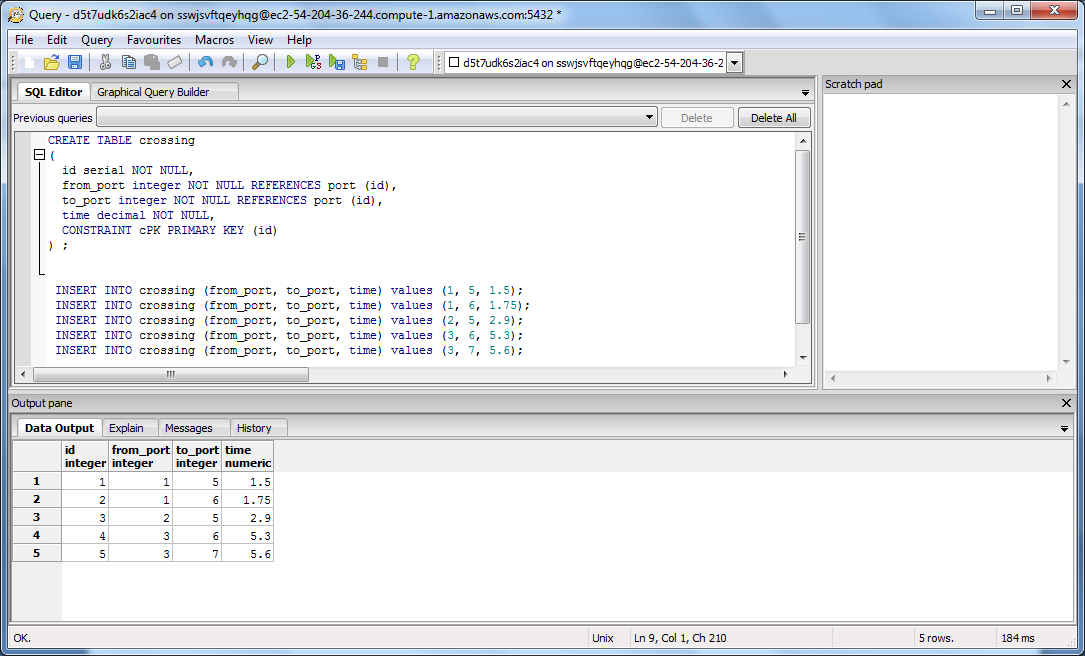
#### pgadmin Tool

PGAdmin Tool allows for administration of local Heroku database but also for remote access Heroku. It allows for a user to execute SQL. A user can create SQL database tables and table columns, insert data into tables and delete data from tables.

Below are a couple of screenshots.



##### Creating the crossing table



## 4.5 org.joda.time

The org.joda.time 2.7 API provides Java application support for dates, times, time zones and durations. JDK 1.8 has a new java.time package. This is similar to the joda.time package, but as I found out to my cost it is not exactly the same. The Android Development Kit uses JDK 1.7 not 1.8, so it is necessary to use the joda.time library and jar files.

I implemented a prototype in JDK 1.8 which required some small changes whwn ported to joda.time.

This project makes use of the Java class org.joda.time.LocalTime. This is a Java time class that represents a specified time in milliseconds. How the Java class will be used in this project is explained as below.

LocalTime.*now()* is a static method to obtain the current time set in the local time zone. The method plusMinutes() makes it easy to add minutes to a LocalTime object [Jod1]. The class org.joda.time.Minutes is also used. The method minutesBetween() calculates the duration in minutes between two LocalTime objects.

org.joda.time.LocalTime:

LocalTime plusMinutes(int minutes)

// Returns a copy of this time plus the specified number of minutes.

org.joda.time.Minutes:

static Minutes minutesBetween(ReadablePartial start, ReadablePartial end)

// Creates a Minutes representing the number of whole minutes between the two // specified partial datetimes.

# 5. Implementation

The application will be deployed to an Android device, a mobile phone for example. When a lorry driver first starts the app the following screenshot will appear on the UI:

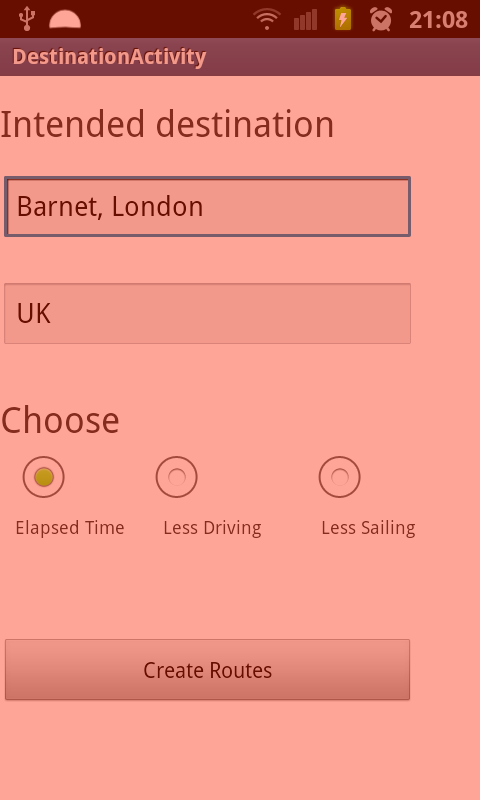


Figure

When the user wants to find the optimal route to his/her intended destination, they must enter the destination data into the app. To start this, they must press the button labelled “Select Destination”.

If they wish to exit the app they will simply press “Exit”.

The UI to enter the destination is presented as below:



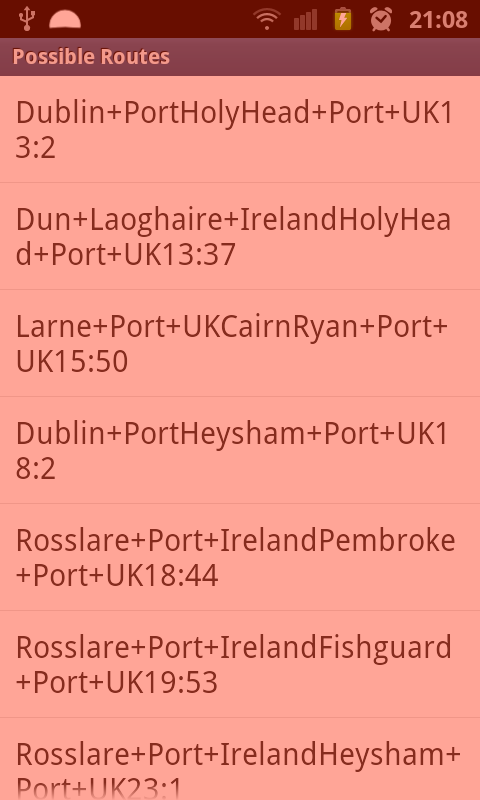
Figure

On the UI, the destination data will be entered into the first textbox beneath the words “Intended Destination”. The user enters the location in the format “Barnet, London” or “Old Trafford, Manchester”. In the second textbox, the lorry driver has to enter the text of the ‘googlename’ of their destination. If they were travelling to the UK they have to type in “UK”.

The driver using the app can also select which metric they would prefer for their optimal route. The three RadioButton widgets beneath the textboxes represent which metric the driver chooses, whether it be “less driving”, “less sailing” and shortest possible “elapsed time”.

To return a list of optimized routes to the user’s destination including road stages and ferry crossings, the user presses the button widget “Create Routes”. The app moves to the next user interface.

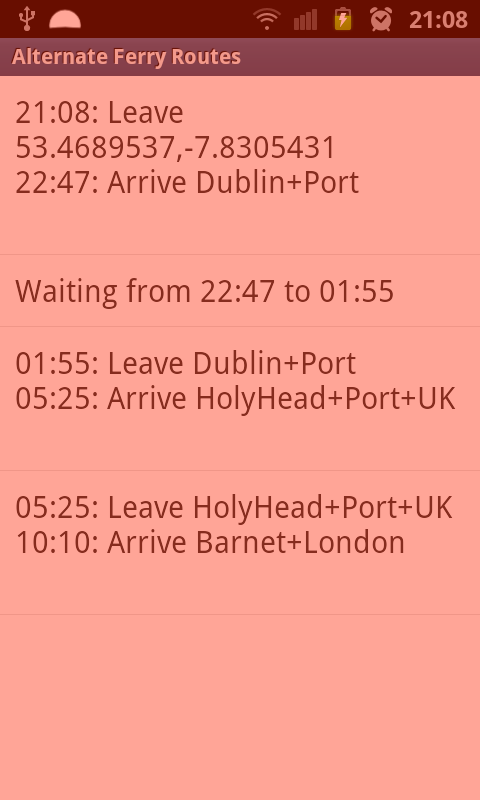
This action returns a list of route from the driver’s current location to the entered destination as displayed below.



Figure

A list of optimal routes is returned. In each component of the list is a listed ferry crossing from a port in Ireland to an arriving port in England as well as the duration it will take the driver to travel from the current location to the destination. Each route is listed in descending order according to the length of time it takes to arrive.

Each route can be selected to display further information. The resulting UI from clicking the first route on the list is displayed below:



Figure

The list seen in the above UI displays the four following stages of the selected route from a user’s current location to the entered destination Barnett, London.

The road stage from current location to the departing ferry port in Dublin

The length of time preparing for the ferry departure

The duration of the ferry crossing

The road stage from the arriving port in Holyhead, Wales to Barnett, London

When the user clicks the first or the fourth widgets, they will be able to see the route highlighted in red on a map overhead. This is exemplified by the screenshot below which highlights the stage of the route from Holyhead port to Barnett, London.



Figure

# 6. Testing

## 6.1. Model Classes & Decision Tree algorithm

Recall the first prototype for the application was the Java implementation. This prototype included the instantiation of the model classes such as Stage, RoadStage, Crossing, Route, Port, LandLocation and Location.

### 6.1.1. Test Cases

For the model classes, the following were all tested:

Constructors (objects were created)

Accessors and mutators (objects were linked together)

toString() method

For the Decision Tree class, the following methods were tested:

buildTree() (the decision tree algorithm is coded)

printRoutes() (the routes returned by the decision tree)

getShortestRoute() (shortest route is returned by app)

## 6.2. Local Postgres Database

This implementation used a local Postgres database used SQL to store and save data concerning the ports, crossings and road stages. Data Access Object classes were written for the database including PortDAO and CrossingDAO.

### 6.2.1. Test cases

The following methods of the DAO classes were tested:

* CrossingDAO.listCrossings(Port from) - (lists all crossings from a particular port)
* PortDAO.listPorts(String country) – (lists all ports from a particular country)

## 6.3. Heroku REST Application

The third implementation moved the data access code into a Heroku REST application. JAX-RS annotations were used to route requests to Java methods. The application accepted requests for information about ports and crossings. |It read the data from the database into model objects. JAXB annotations serialized the objects into XML.

### 6.3.1. Testing

All testing was done using a browser. The browser sent GET Requests to the application which returned ports and crossings data in XML format.

### 6.3.2. Regression Testing

* The model classes were changed and JAXB annotations were added to the classes. This determined how the classes were serialized into XML. The annotations were tested.
* The data access object classes were to a large extent unchanged. A new UtilDAO.getConnection method was written to connect to the Heroku database and was tested.

### 6.3.2. Test Cases

The resource classes, PortService and CrossingService were written. The following methods were implemented and tested using a browser.

* PortService.getPortsByCountry()
* CrossingService.getCrossingsbyPort()

## 6.4. Java Client parsing the XML

The client side Java application now sends REST requests to the Heroku application and reads the XML back. It parses the XML and creates new instances of the model objects.

## 6.4.1. Test Cases

On the client side the following classes were written. The DMService and PCService provide access to the distance matrix service and the ports and crossings service. For parsing the data ParsePorts, ParseCrossings and ParseRoadStages were written. The following methods were implemented and tested:

* DMService.getStages() (returns a list of road stages from a single location to multiple ports)
* PCService.getCrossingsbyPort() (returns a list of available crossings from a departing port)
* PCService.getNextCrossing(List<Crossing> crossings, int destinationPort, LocalTime time) (returns the next available crossing after a specified time)

Two other test classes were written where the current location and destination were explicitly hard-coded:

* TestDistanceMatrix
* TestPortsCrossings

These were used to test access to services and also the parsing of the returned data.

### 6.4.2. Regression Testing

The DecisionTree algorithm was now retested to ensure that it worked with the objects created as a result of service calls.

### 6.5. Android Client

The Java client was ported to Android. Calls are now made both to the Heroku application and the Google Maps API. Screens were created as shown above. The application was repeatedly tested and the output examined for correctness.

# 7. Conclusion

The primary objective of this project was to build an application to be used by lorry drivers. The app should return to the user a list of optimal routes including ferry crossings for the driver to take in order to arrive at their intended destination. If a driver were to miss their ferry due to weather or bad traffic or if the ferry had been cancelled, the driver could use the app to rearrange their journey. The app had to be easy to use. The user would simply click on the app and enter their destination data. The app also had to be portable. It could be installed on a user’s mobile phone.

The application consisted of client and server side components. The client application was designed as an Android application. The server side of the application was implemented as a REST service and deployed to the cloud computing service Heroku and used the Postgres database. The ports and crossing data was taken from the Derry Bros website and inserted into database tables in Postgres. The REST service, implemented in Java uses JAX-RS for routing HTTP requests to Java methods. These methods accessed the database and JAXB was then be used to serialize data to XML. The XML is returned to the client in a HTTP response.

As well as accessing the server side component, the client also accesses the Google Maps and Google Distance Matrix APIs. It retrieves road route data from the Google Distance Matrix API. Google Maps allows for highlighting selected routes on map overheads. To optimize the alternate routes for the application, a decision tree was implemented in the client side of the application. Once the algorithm was applied all possible routes were found and sorted in descending order according to how long the route would take to travel. The app also allows for other optimization metrics which for example weighted travel times on roads higher than ferry crossing time or vice versa. The visitor pattern is used to insert these different metrics.

Regarding the initial requirements, these were to a large extent met by the project. In my opinion, the application works very well. It is easy to use. It finds possible routes and orders them correctly. The routes are highlighted nicely on maps providing a clear user interface for the lorry drivers. With some modifications the application could be used by a logistics company such as Derry Bros to provide a good service to its clients.

The following work would have to be carried out if the application were to be used by a company such as Derry Bros.

* Some mechanism for caching frequently used destinations, for example drop-down options for the textbox where the user inputted their intended destination instead of typing in the entire location.
* The application would need to be extended to deal with routes which contained more than one ferry crossing. This would support routes for example from Ireland to England and on to the continent.
* The server side would have to be integrated with the Derry Bros website and obtain port and crossing information from the same database. The only changes needed would to the data access object layer in the server side component of the application.
* Additional optimization metrics would probably be required. This would need further discussion with the lorry drivers and staff in Derry Bros. The implementation of these metrics would be made easier by the fact I have used the visitor pattern in the application. All is required is to provide another that implements the CostMetric interface.

I will now evaluate the design decisions made in the project. Android was the chosen mobile platform for the application. It was taught in a college course and I had some experience with using it on Coop, so it was the obvious choice. REST was used for client-server communication. Using the REST architecture was also an obvious choice as it too is very popular. Heroku was chosen for the cloud computing infrastructure. This is a Platform as a Service (PaaS) and it provides a J2EE stack. It has very good documentation and provides a very understandable REST server example. This helped a lot in getting the server component running on Heroku. MySQL is by far the most popular open source relational database. However Postgres is the default database for Heroku, so it was chosen. The pgAdmin tool for Postgres was easy to use, and made it efficient to create database tables and to insert data into these tables. It worked equally well on a local database and on the remote Heroku database. JAX-RS was used for implementing the REST service. It is a Java standard for REST. JAXB was used to serialize data to XML format. Very little (no) programming is required, just a couple of JAXB annotations on the domain classes. Good tutorials for both JAX-RS and JAXB are available on <http://www.vogella.com/tutorials/>.

On the client side of the application, the decision tree was chosen for the optimization of routes. The decision tree did exactly what it was required to do and worked rather well. It was very possible due to the fairly small number of possible routes. The XmlPullParser was used to parse the XML Port and Crossing data. It is the recommended XML parser for Android, so again it was an obvious choice. I had some experience using this XML parser during my Coop work placement. The Google Distance Matrix and Google Maps API were standard choices especially since I was using Android.

The main problems I encountered were the following. During the development of the Java prototype, I used the package JDK 1.8. In order to process the time instances in the project application I used the java.time package. When I came to deploy the Java client to Android, I found that the Android libraries don’t contain the java.time package. To get around this, I included the Joda Time library which I thought was compatible with JDK 1.8. However, while it was largely similar it was not a 100 % compatible and some rewriting of the code was necessary.

Secondly, the instance where I had to draw a route on a map, the API required used JSON rather than XML. So I had no experience in parsing JSON, but I found code on an online tutorial which I used directly in my project. [Map1]

I believe this was a good Final Year Project for the following reasons. First, thanks to Dr. Enda Fallon in the SRI, I was given access to a real world problem to see what sort of solution I could come up with. The problem was fairly well defined and following discussions I was able to identify a set of requirements. These requirements were mostly met as described above. During the project I was able to gain further exposure to technologies I had used on my Coop placement, namely Android, and accessing REST services from Android using XML Pullparser, and HTTP. I also got to work with new technologies such as JAX-RS, JAXB, Heroku, Postgres and implementing REST service in Java. I had used Google Maps before on my Coop. I also got to use Google Distance Matrix API and learnt how to draw maps on Google Maps. I also got some exposure to Git and Maven. All of the above I believe are important skills for a Java developer.

Most importantly, I got to work on a real world problem and found a solution to that problem. In order to do that, I had to carry out research in a number of new areas. I learnt that with good documentation and examples, it is possible to start working with such new technologies. This has given me great confidence as a Java developer.

# 8. References

[And1] – Parsing XML Data, [online] available: <http://developer.android.com/training/basics/network-ops/xml.html> [accessed 15 Nov 2014]

[Git1] - Git Documentation, [online] available: <http://git-scm.com/doc> [accessed 18 Nov 2014]

[Gog1] - The Google Distance Matrix API, [online] available: <https://developers.google.com/maps/documentation/distancematrix/> [accessed 28 Feb 2015].

[Gog2] - Google Maps Android API v2, [online] available: <https://developers.google.com/maps/documentation/android/> [accessed 28 Feb 2015]

[Her1] - Getting Started with Java on Heroku, [online] available: <https://devcenter.heroku.com/articles/getting-started-with-java#introduction> [accessed 25 Nov 2014]

[Her2] - Heroku - Eclipse Integration, [online] available: <http://eclipse-plugin.herokuapp.com/> [accessed 25 Nov 2014]

[Her3] - Heroku Postgres, [online] available: <https://devcenter.heroku.com/articles/heroku-postgresql> [accessed Nov 29 2014]

[Her4] - Deploying with Git, [online] available: <https://devcenter.heroku.com/articles/git> [accessed 19 Nov 2014]

[Jod1] – Class LocalTime, [online] available: <http://www.joda.org/joda-time/apidocs/> [accessed 29 Nov 2014]

[Mal1] - Sales and Chips, Feature Column, Monthly essays on mathematical topics, Joe Malkavitch, [online] available: <http://www.ams.org/samplings/feature-column/fcarc-tsp> [accessed 05 Jan 2015]

[Map1] - Draw Path on Google Maps Android API, [online] available: <http://javapapers.com/android/draw-path-on-google-maps-android-api/> [accessed 03 Jan 2015]

[NEO1] – Vehicle Routing Problem, Networking and Emerging Optimization, [online] available: <http://neo.lcc.uma.es/vrp/vehicle-routing-problem/> [accessed 03 Nov 2014]

[Pos1] - PostgreSQL 9.4.0 Documentation, [online] available: <http://www.postgresql.org/docs/9.4/interactive/index.html> [accessed 17 Dec 2014]

[Til1] - A Brief Introduction to REST, Stefan Tilkov, [online] available: <http://www.infoq.com/articles/rest-introduction> [accessed 05 Nov 2014]

[Vog1] - REST with Java (JAX-RS) using Jersey – Tutorial, Lars Vogel, [online] available: <http://www.vogella.com/tutorials/REST/article.html> [accessed 05 Nov 2014]

[Vog2] - JAXB Tutorial, Lars Vogel, [online] available: <http://www.vogella.com/tutorials/JAXB/article.html> [accessed 11 Nov 2014]

[Vog3] - Android XML Processing with the XmlPullParser – Tutorial, Lars Vogel, <http://www.vogella.com/tutorials/REST/article.html> [accessed 27 Nov 2014]

[Vis1] – Visitor Pattern|Object Oriented Design, [online] available: <http://www.oodesign.com/visitor-pattern.html> [accessed 05 Mar 2015]