# CAR INSURANCE SYSTEM REPORT
# ALEX ROAN

## INTRODUCTION

Representation State Transfer, or REST, is defined as a stateless architecture that enhances the interactions between clients and services by having a very limited number of operations. By assigning resources unique URIs, REST enhances the flexibility of such interactions and reduces ambiguity by constraining the functions to four specific actions: GET, POST, PUT and DELETE.

The Car Insurance System I have developed in this project takes advantage of Ruby on Rails' RESTful features and is able to interact with brokers outside of its server RESTfully. My Broker application runs using Javascript and the JQuery library to interact with the underwriter application.

## UNDERWRITER APPLICATION ARCHITECTURE

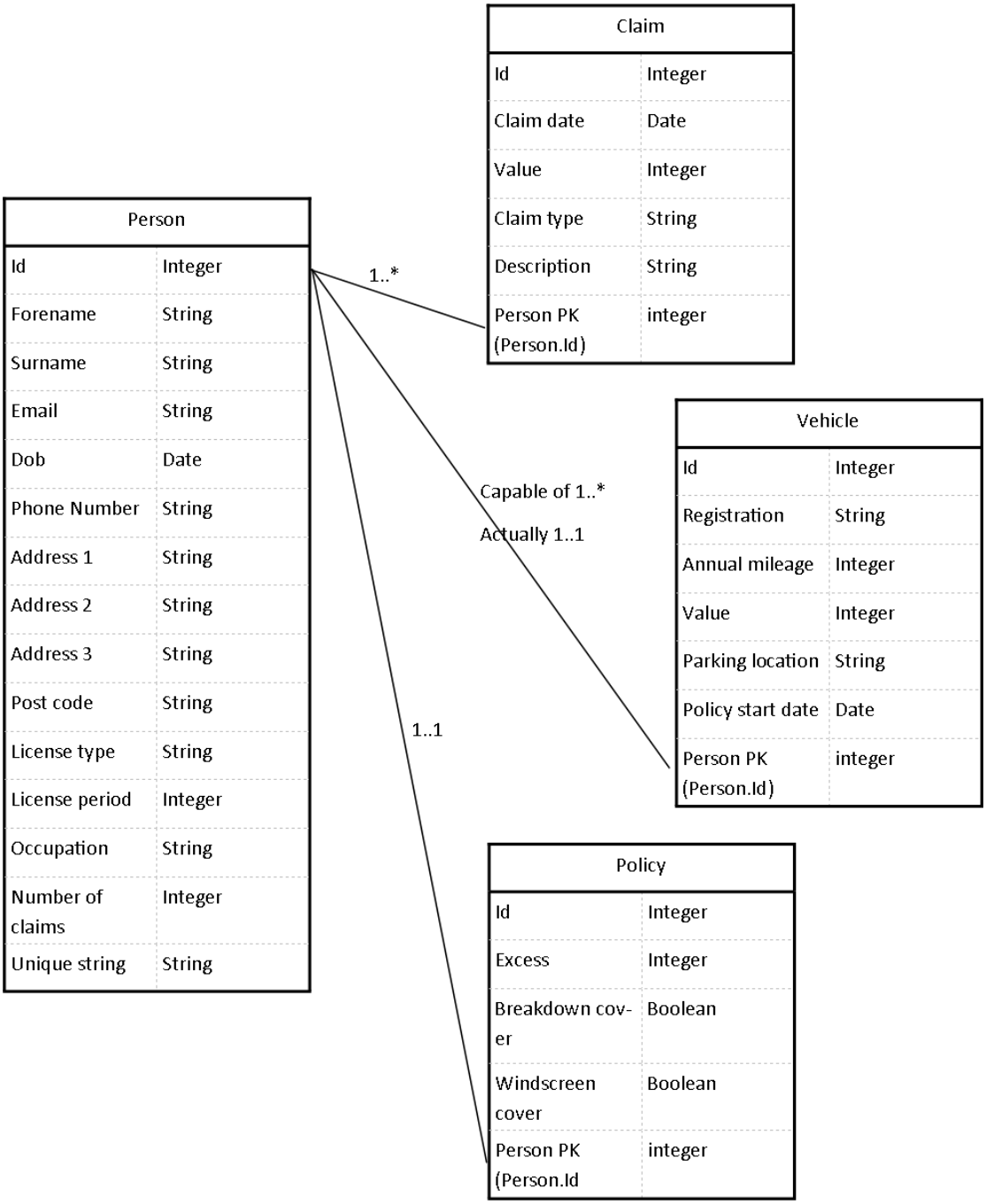
My underwriter application is comprised of 4 main tables which represent all of the information needed to store and manage a user's details regarding car insurance. The tables are: Person, Claim, Vehicle and Policy.

The Person table stores the user's title, forename, surname, email, date of birth, telephone number, address details, license type, license period occupation, number of claims in the past 3 years and a randomly generated string. The address details are split into four separate fields: Address 1, 2, 3 and Post Code. All but telephone number address 2 and address 3 are compulsory fields, and the randomly generated string is generated upon adding a record to the Person table.

The Claim table stores the claim date, value, type, description and the primary key (id) of the related Person table entry for each claim the person has made in the past 3 years. All but the claim description fields are compulsory.

The Vehicle table stores the registration, annual mileage, value, parking location, policy start date and the primary key of the related Person table entry. All fields in this table are compulsory.

The Policy table stores the excess, breakdown cover, windscreen cover and the primary key of the related entry in the Person table. All fields in this table are compulsory, with the excess field being generated when the user has filled all details out.



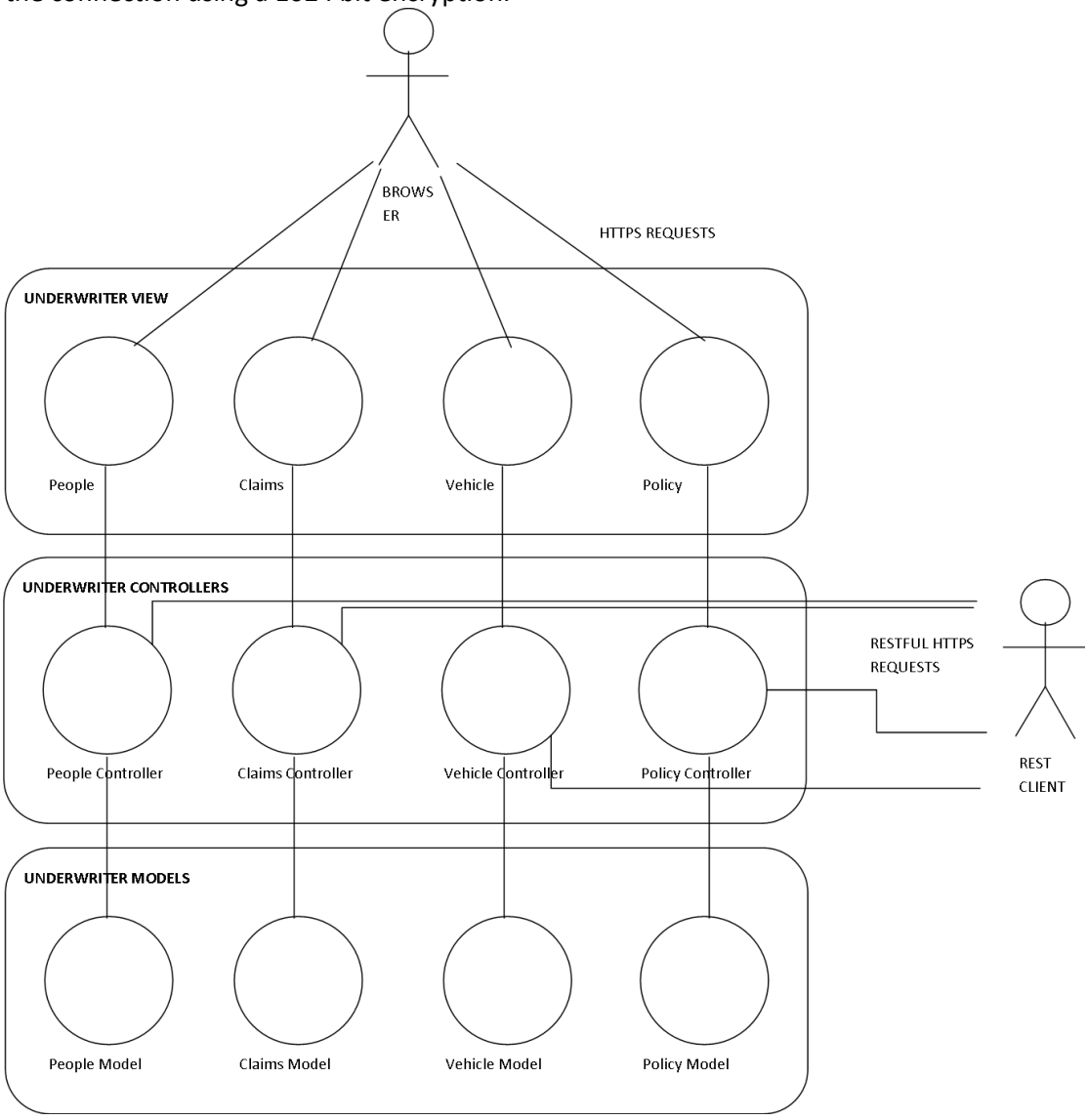

In my opinion according to the specification and in the context of the problem it poses; the Person, Vehicle and Policy tables could have been merged into one due to the one to one nature of the required data storage method. However, this would not adequately represent the real world where one person can own multiple vehicles and take out multiple policies on these vehicles, which is why I have designed the database as if the relationships between these three tables could be one to many rather than one to one.

If the broker application enabled users to take out multiple policies on multiple vehicles, I would edit the Policy table to include a Vehicle PK field which linked to the corresponding vehicle for that policy. But as it stands there is no real need for it because everything links back to one Person object.

When directly accessing the underwriter application from a browser, the tables can be viewed page by page as per the MVC pattern of Rails. I've included pagination in the views for each of the tables in the database so they are more easily read via this method.

This was done by including a pagination gem within the gem file and modifying views so that the correct items on each page would show.

The Underwriter application takes advantage of HTTPS for receiving requests from browsers or broker applications by being run using the command “bundle exec ruby bin/secure_rails s”. The “bin/secure_rails” file takes advantage of an RSA certificate to encrypt the connection using a 1024 bit encryption.



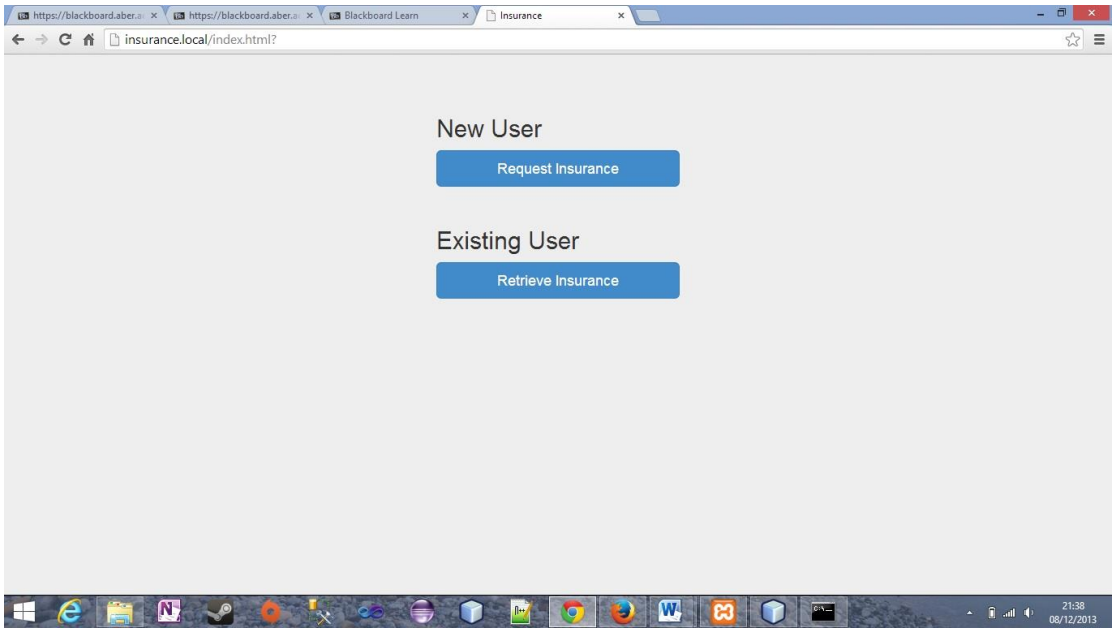

## UNDERWRITER VALIDATION

For each of the models I have included validation for a number of fields. The fields which are required to be not null are validated for their presence, integer fields are validated for their numericality more than or equal to zero and emails are checked for their format and uniqueness within the database. Validating these fields in the model classes enables strict rules for what is entered into the database upon request.

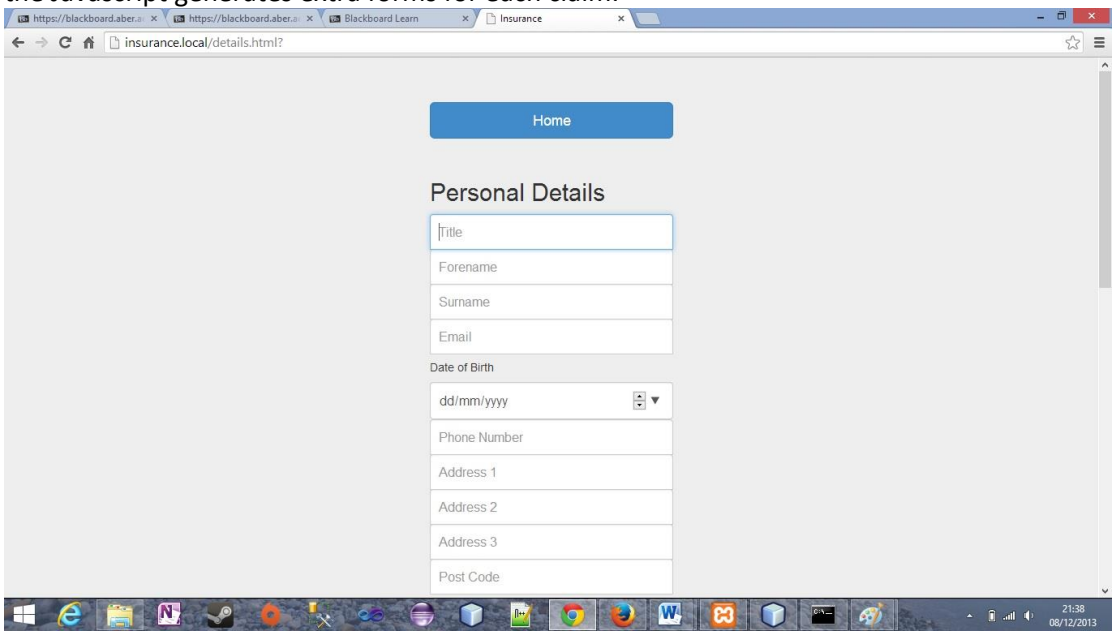
# RESTful Broker Client Architecture

My restful broker client is comprised of html pages and Javascript files which deal with the RESTful communication between the broker and the underwriter as well as providing features on the broker pages. The Javascript files validate, verify and gather data entered by the user and make use of JQuery Ajax functions to deal with POST and GET requests to and from the underwriter.

The Broker site is structured into three pages. The homepage is a simple selection between two options for the user: for new users “Request Insurance”, and for existing users “Retrieve Insurance”.

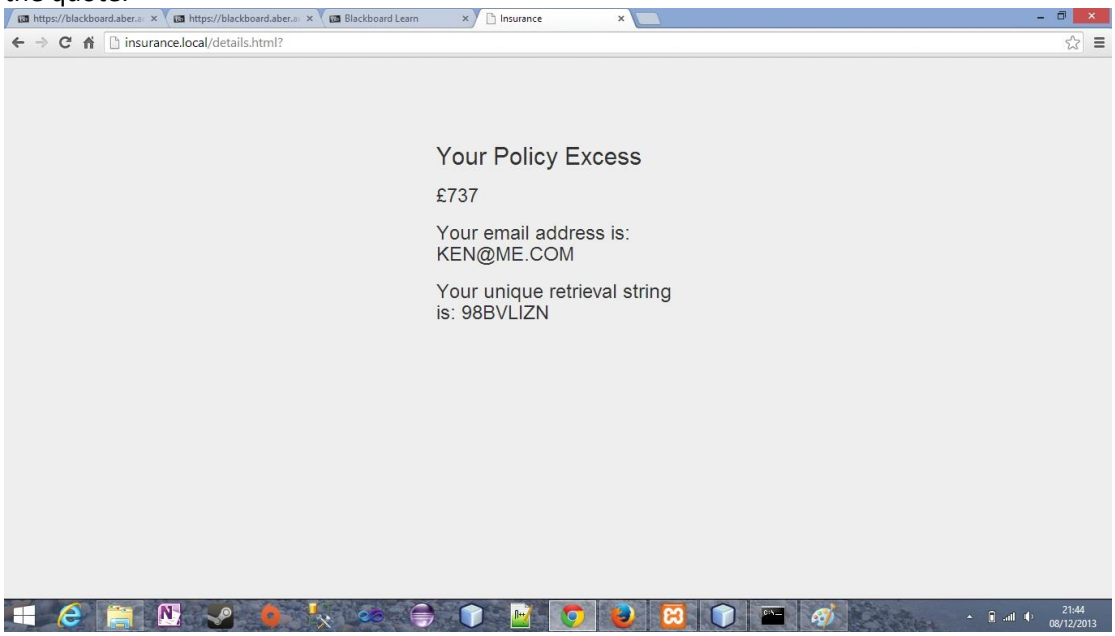



If the user chooses to request insurance they are taken to the details.html page where they fill in their personal, claim, vehicular and desired policy details. The page does not have any claim forms upon load but depending on the amount of claims the user enters, the Javascript generates extra forms for each claim.




Once the user has filled in the forms on the page, the submit button at the bottom of the page starts validating all of the fields for any unfilled or incorrectly filled inputs. Only when all required fields are filled correctly will the broker client begin to send POST data to

the underwriter. The email field is unique for each user in the underwriter application so if a user enters in an email address that is already being used they will not be able to request insurance on this page. If all the data is successfully saved on the underwriter application then the user is displayed their quote, their email address and their identifier string which is a randomly generated string on the underwriter server. This will be used for later retrieval of the quote.




If an existing user chooses to retrieve a quote then they are taken to the retrieve.html page. This prompts them only for their email address and identifier string generated when they requested the quote. When the user submits their data the form is validated and sent to the underwriter. If the email and identifier match the records in the underwriter database then the corresponding quote and policy details are displayed, otherwise a warning is displayed outlining the fact that the details entered do not match the underwriter's database. At all times the user has the option to go back to the home page using the large "Home" button at the top of every page (other than home itself).

## JQUERY AJAX

I used the Javascript JQuery library to transmit data to and from the underwriter application. It enabled me to send data via HTTPS and receive responses which could be dealt with easily. When the functions to start posting data to the underwriter were called however, I observed that Javascript and AJAX often didn't wait for a response before continuing processing the code immediately following. In development, methods which used POST to send data to the underwriter would return before actually receiving data back, meaning the following conditional statement to continue posting further information would be false. The way I worked around this was to write a chain reaction like process, in which the following POST method was called from the previous one, instead of the previous return some value to a higher level method and that method passing it to the next.

## BOOTSTRAP

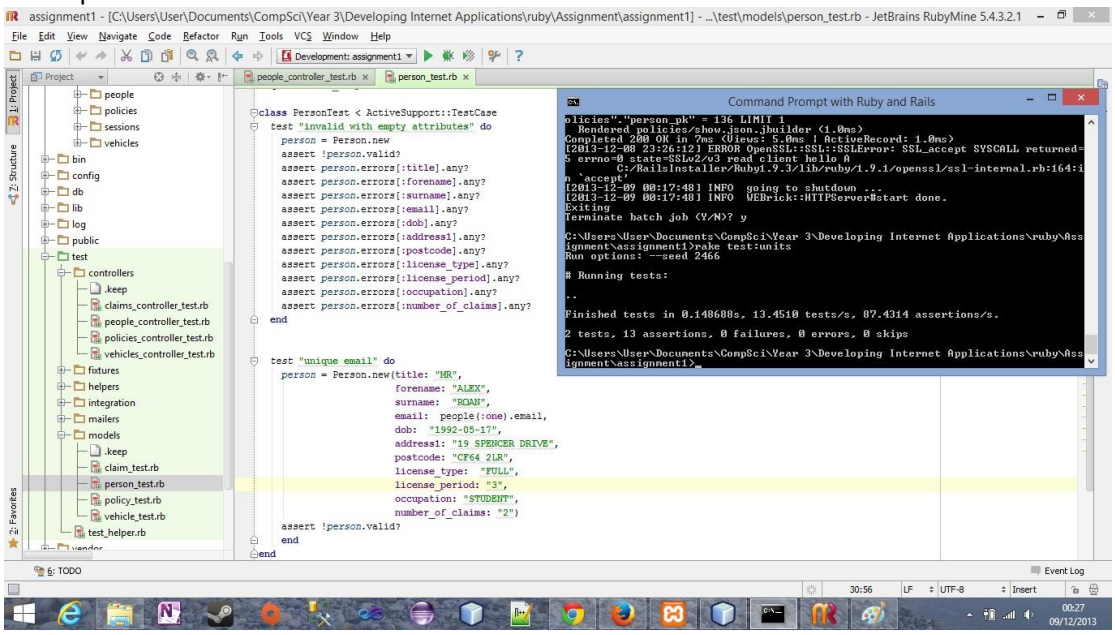
I used a framework called bootstrap for the front- end of my broker application. It provides a powerful method for easier and faster web development. Bootstrap is comprised of a number of CSS and Javascript files which make designing web pages far easier than the traditional approach of writing all parts of the website.

## REST FORMAT

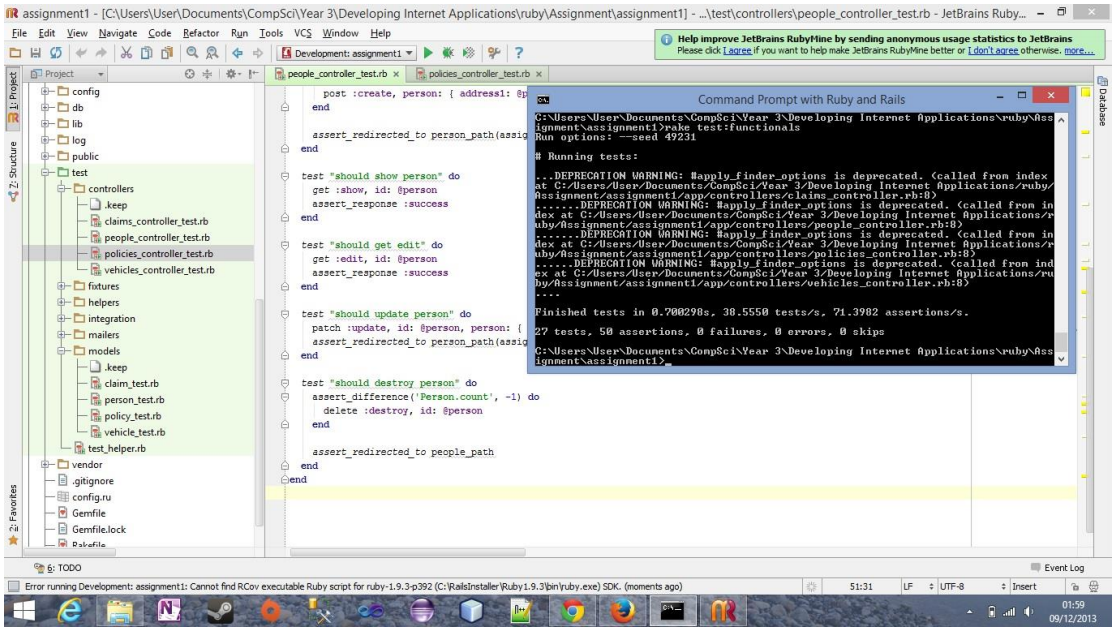
The requests that are sent between Broker application and underwriter application are in JSON format. I chose this format because it is extremely easy to deal with within Javascript and provides a lightweight mechanism for transferring structured data. I had toyed with the idea of using XML because it is supported as fully as JSON is in Rails. However, it is slightly more difficult to perform the parsing tasks in the Javascript run Broker application with XML than it is with JSON.

# TESTING UNDERWRITER

I have performed unit tests on a few of my models and controllers in Rails as shown in the photo below.




As well as unit tests, functional tests were set up and run on the models and controllers.



# TESTING BROKER AND UNDERWRITER TOGETHER – FUNCTIONAL REQUIREMENTS

| ID | Requirement | Description | Inputs | Expected Outputs | Pass/Fail | Comments |
|---|---|---|---|---|---|---|
| 1 | FR1 | Request Quotation | Enter values for all fields in correct formats for a brand new customer(email in correct format, currency etc) | New quote with a new identifier for later retrieval | P | |
| | | | Enter no values for any of the fields | No quote qill be requested and message pop up explaining that fields were missing | P | |
| | | | Enter all values correctly but with an email address already in the database | Message appearing informing that the email already exists and the user is not allowed to submit | P | |
| | | | Enter an Incorrectly formatted email address | Message prompting incorrect email format and user not allowed to proceed | P | |
| 2 | FR2 | Save a Quotation | First step in FR1 | Data correctly saving itself to the database | P | |
| | | | User email already exists | Data not stored and message | P | |

| | | | | | | |
|---|---|---|---|---|---|---|
| | | | | relayed explaining why | | |
| 3 | FR3 | Retrieve a saved quotation | User enters correctly matching email and identifier string | Data retrieved correctly | P | |
| | | | User enters falsely matching email and identifier string | No data retrieved and message displayed explaining why | P | |
| | | | User enters no data | Message appear displaying the missing data fields | P | |

# EVALUATION

My method of approaching this assignment was decided upon early on and stuck throughout the majority of developing the solution except for when newly discovered and unexpected dilemma's caused me to change course (such as AJAX not waiting for a response before moving on the next line of code). However, I feel after I grew into the project and started to discover more and Ruby on Rails and the tools it provides to create large scale applications, I would have changed the methods in which I went about creating the solution.

I had designed my routes.rb file fairly early on in the process, and had decided that the only approach to take when sending and receiving data back and forth the underwriter was to do so in chucks, table by table. In my project, the broker application sends the new Person data, and upon retrieval of a response (as long as the response is good) calls the next set of data to be POSTed to the underwriter claim by claim and so on. I discovered late on that a much larger range of parameters in JSON could have been send in one large chunk to the underwriter application, and could be dealt with and dispersed amongst the controllers within Rails to store the data to each table. This is how I envisage redesigning the RESTful interaction if I had the chance to re-do this project.

I found the Javascript aspect of the broker application somewhat fiddly, and had considered using another language such as PHP as a more suitable language for it. However, I could not find a library that provides all the features for RESTful interaction which includes all four of the POST, PUT, GET and DELETE.

## SELF-EVALUATION

I found adapting to the concept of a dynamically typed language quite difficult to begin with. The logic of it was something quite different to the traditional statically typed languages I have learnt and used in the past and initially I found it somewhat counter-intuitive. However, I can now see the advantages of dynamically typed languages and what they offer.

Ruby on Rails was difficult to get to grips with to begin with. Trying to decipher what was causing error messages proved to be quite challenging I've learnt that it does have some very powerful features if used correctly.

If I were to do this project again I would include a user authentication module in which the user would register their details before being able to request a quote. The service would email a verification passcode to the user upon account creation. I would also put more time into developing the underwriter in Rails so that instead of taking multiple POST requests at a time for each of the tables, it would be able to handle a POST request which has all of the database data in one. The Person, Claims, Vehicular and Policy data all as JSON parameters and dealt with by Rails.

I feel I should be awarded between 60% and 65%. I believe that I have fulfilled all of the functional requirements: being able to request, save and retrieve a quote from the underwriter using the Broker application. In doing this I have implemented a RESTful interaction between Broker and underwriter using HTTPS as the protocol of request. The underwriter is also in itself a fully functioning website. I have implemented Unit and Functional tests in Rails and produced a test table for analysis of the two applications working together. Where I fall down is on the advancement of some of the features I could have used within rails for the underwriter application.