SE31520: Car Insurance System

Department of Computer Science, University of Aberystwyth

Monday, December 9, 2013

Diana Silvia Teodorescu

dst1@aber.ac.uk

Table of Contents

[Introduction 3](#_Toc374311927)

[System Architecture 3](#_Toc374311928)

[Technologies 3](#_Toc374311929)

[Architecture & Design Diagrams 3](#_Toc374311930)

[REST API 5](#_Toc374311931)

[Changes & Rationale 6](#_Toc374311932)

[RESTful broker client 7](#_Toc374311933)

[Technologies 7](#_Toc374311934)

[Architecture & Design Diagrams 7](#_Toc374311935)

[RESTful Interoperability 8](#_Toc374311936)

[Rationale 8](#_Toc374311937)

[Resultant System 9](#_Toc374311938)

[Testing 10](#_Toc374311939)

[Strategy 10](#_Toc374311940)

[RESTful Client Testing 10](#_Toc374311941)

[Results 11](#_Toc374311942)

[Testing screenshots 13](#_Toc374311943)

[Self Evaluation 16](#_Toc374311944)

[Approach 16](#_Toc374311945)

[Problems & Compromises 16](#_Toc374311946)

[Areas of Expansion 17](#_Toc374311947)

[Output & Assessment 18](#_Toc374311948)

[Acknowledgements 18](#_Toc374311949)

# 

# Introduction

The application presented is an Online insurance broker that offers an insurance quote dependent on the information provided by a customer in an online form.

The Insurance Broker web application can be used to send information to a rails server in a secure way and retrieve saved information at a later date. The data transfer is RESTful by using web technologies such as Javascript and Ajax.

This document will present the RESTful relation between the web broker and it's underwriter - the rails database model, and it is a description of the programming stages that resulted in the application and an overview of the implementation decisions and compromises.

# System Architecture

### Technologies

The underwriter system was created with Ruby on Rails basic installation. It uses Rails version 4.0 and Ruby version 1.9.

The Rails database has been generated through the Ruby command prompt and it resulted in providing most of the methods and controllers necessary to build the tables needed.

The basic gems that were installed with the rails version and were used for this type of system are:

gem 'rails'

gem 'sqlite3'

gem 'sass-rails'

gem 'uglifier'

gem 'coffee-rails'

gem 'jquery-rails'

gem 'turbolinks'

gem 'jbuilder'

To use HTTPS a certificate has been generated using OpenSSL and implemented in the rails project using WEBrick

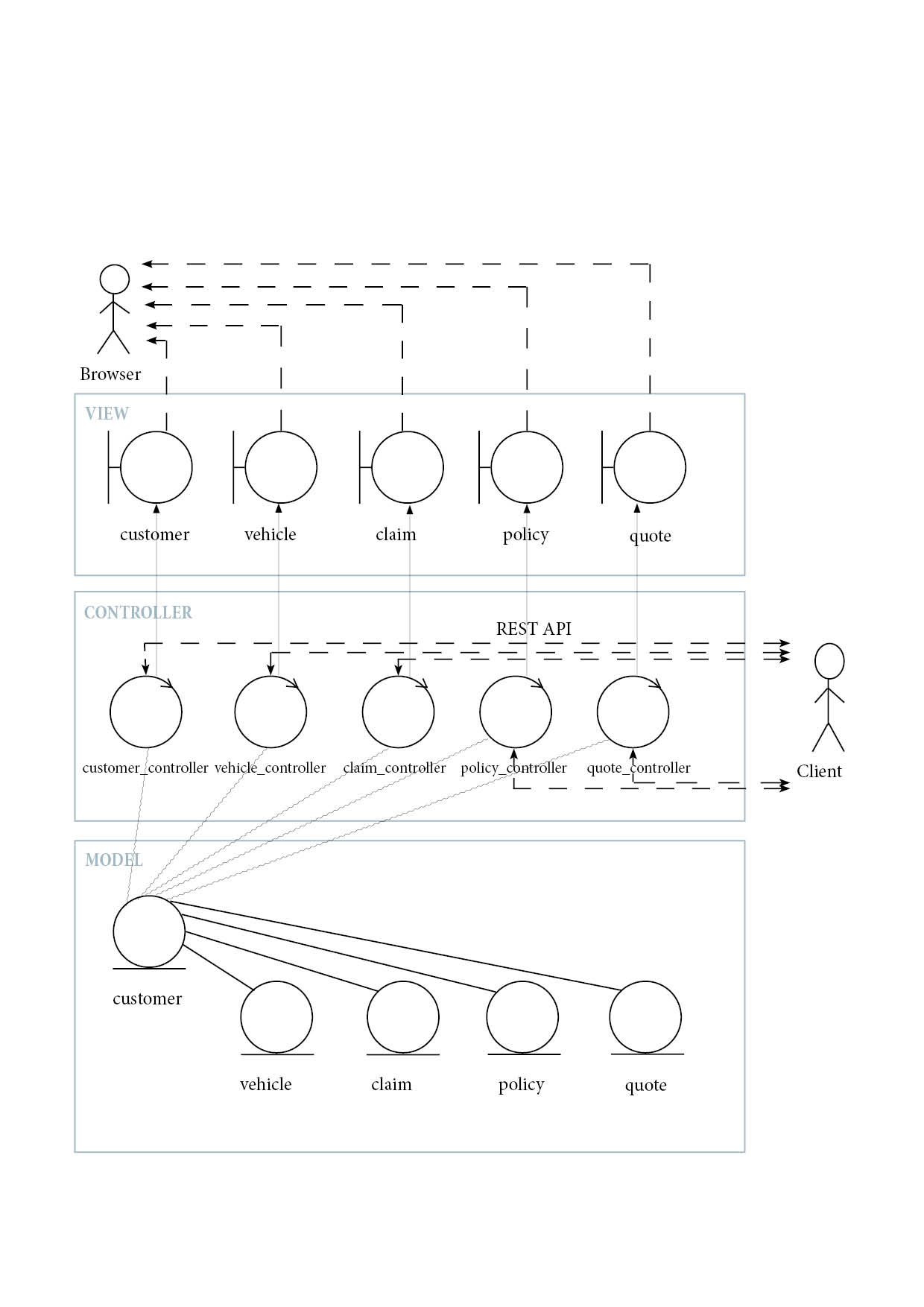
### Architecture & Design Diagrams

The architecture of the system has the default Rails structure. This means the application underwriter is using the Module View Controller pattern that divides the system in three components that interact with each other.

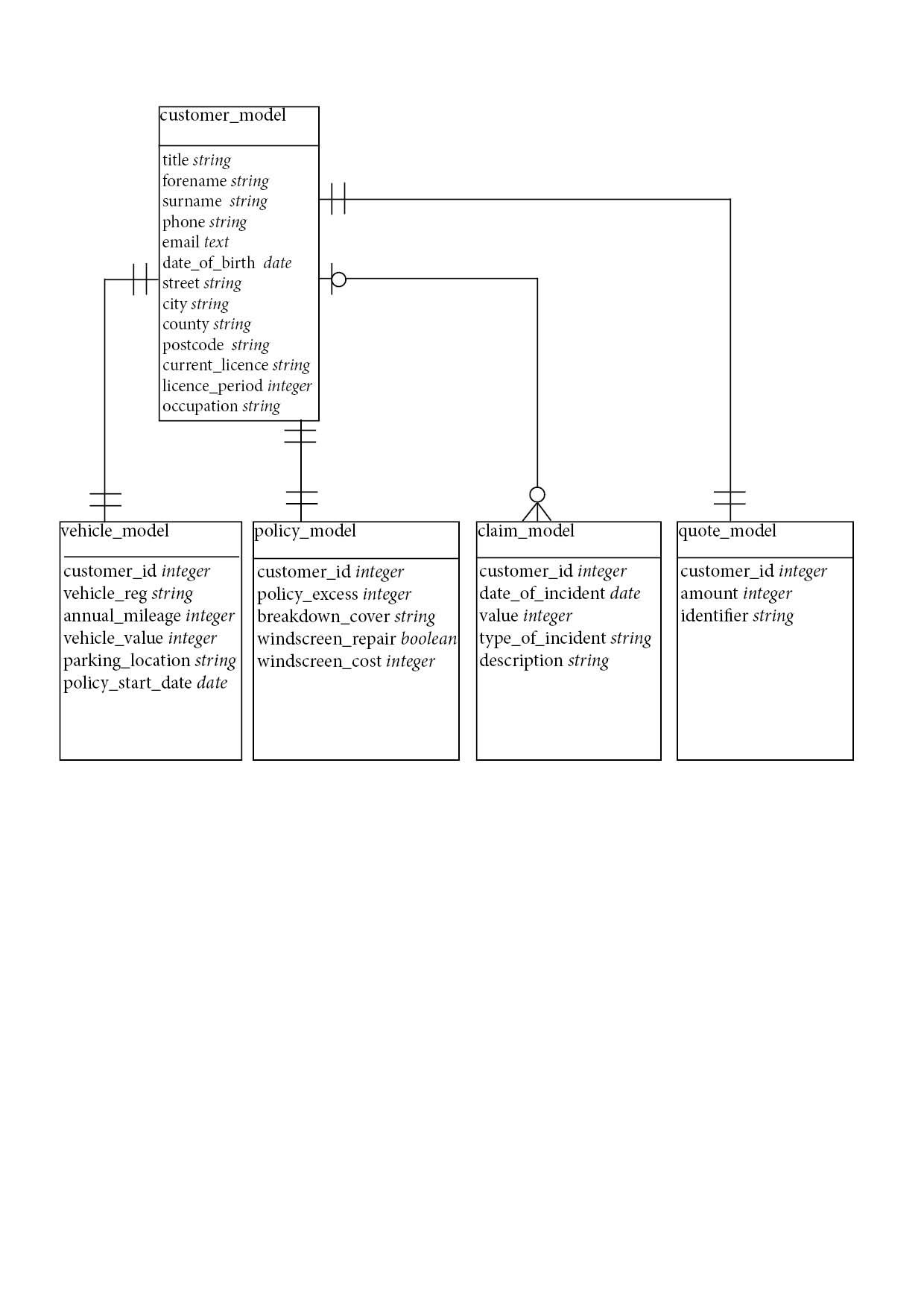
The diagram shows the structure of the system and the relationships between its objects. There were changes made relating to the associations between the models created such as being able to hold multiple claims for a client. Also the system connects to an eternal Client through a RESTful architecture.

The views of the Rails system were mostly used during testing and understanding of the architecture but they aren't in the foreground of this project and were left mostly untouched.

The following diagram shows the Model View Controller system and it's interaction with the other entities. The View can be accessed through a browser and the User Interface is connected to the system through the RESTful API and Controllers.



The following image shows the entity diagram in the Model part of the system and the one to many and many to many relationships between the tables, including details about their variables.

**

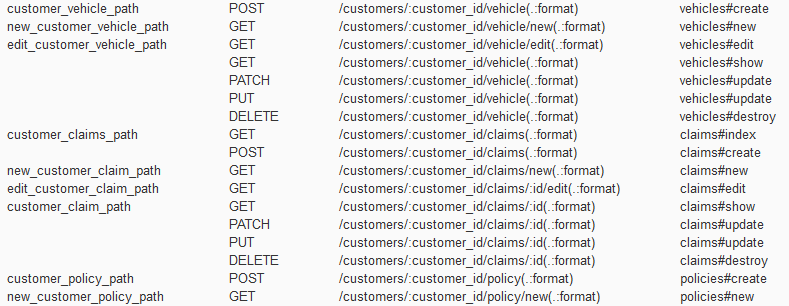
### REST API

The transmission of data between the Rails server and the Client is designed so it's independent one from the other allowing the portability of the system.

For this it was important to change the routes of the system to be able to send and retrieve the information to/from the model via POST and GET.

The following image shows the routes for the customer and its relations with the other models. In case of claims there is not only the customer\_id in the link but also a claim ID because of the has many relation. All the other models can be accessed through a customer with a specific ID.

The following image shows part of the system paths. It can be observed the relations between the customer and vehicle as a has one relationship and between customer and claim as a has many relationship.



### Changes & Rationale

The module chosen in this case were: Customer, Vehicle, Policy, Claims and Quotes. This choice was based on the first functional requirement regarding the process of requesting a quotation. The information required from the user will be stored depending on subject in different data tables connected to each other.

Besides for the four objects extracted from the FR1 I chose to add a fifth table containing the result of the query: amount and quotation identifier.

**Rails Associations**

Te connection between all models is made through the customer\_id which acts as a primary key in the Customer Model and as a foreign key in all the others. The interaction between the models was created by changing the routes of the application and the model files.

When using the web application the user can request the insurance only for one vehicle with one policy but it can have multiple past claims. This meant having different relationships present in the customer\_model:

has\_one :vehicle, dependent: :destroy

has\_many :claims, dependent: :destroy

has\_one :policy, dependent: :destroy

has\_one :quote, dependent: :destroy

The dependent destroy is used to have the facility to remove all data of one customer from each of the tables related to it.

**RESTful API**

To be able to use the RESTful API some changes needed to be implemented in the Application\_controller.

*protect\_from\_forgery with: :null\_session* - this removed the extra protection for CSRF vulnerabilities.

*def set\_access\_control\_headers*

*headers['Access-Control-Allow-Origin'] = '\*'*

*headers['Access-Control-Request-Method']='\*'*

*headers['Access-Control-Allow-Headers']='\*'*

this allows to send and receive XHR/ XMLHttpRequest requests to and from the Rails server. Otherwise the system would prevent a client side application from running with the underwriter. If we knew the server that will make the request we could add a rule here in the *headers['Access-Control-Allow-Origin']* to be set to the specific url of the website that will hold the Broker

The show.json.jbuilder files were altered to be able to show the foreign key: the customer\_id. This was also added to the parameter lists for each model.

**Generating quote and identifier**

After data is sent from the Web Broker the quote\_controller creates the amount and the quote that would be presented to the User on a separate page. To create the two variables I have used:

- for amount: a random number between 10 and 1000

- for identifier: a random string connected with the Customer ID - this makes it less likely to get two identical identifiers. For the random string I have used *SecureRandom.hex(2)* that generates a random string of size 2x2 composed from letters and numbers.

**Retrieving a previous quote**

The third functional requirement requested the option of retrieving data from the database depending on two parameters: an identifier and the customer email. For this purpose we needed to create a separate method and route that would obtain the data provided and comparing it to the one previously entered. This was possible by adding a new route that can take the information provided by the client and send it to a new method, in this case called get\_customer\_details.

*match "quote/:email/:identifier" => "quotes#get\_customer\_details", :constraints => { :email => /.+@.+\..\*/ }, via: [:get]*

*S:\compsc\dst1_Assigment_SE31520\email_ident.png*

The function is called from the quote controller and it compares the *id* of the customer with the email provided with the *customer\_id* of the quote found by the *identifier* provided. If these are the same then the data is send in JSON format to the REST Client.

The constraints are used both to allow access to a link that contains unpermitted characters such as @ and . and also for security. The link can be accessed only if there is an email address.

# RESTful broker client

### Technologies

The RESTful broker client is written in a combination of html, css, javascript and ajax. The form takes each input and stores it into an array of json objects such as: vehicle, customer, claim, quote.

Ajax is used to connect to the Rails server and POST or GET the information necessary. To understand how to run and use the entire system, online broker together with the underwriter a README.txt file and a screencast has been provided.

### Architecture & Design Diagrams

The Online Broker stands independent from the Rails MVC and it is implemented remotely.

The system is constructed from different HTML pages connected through style, scripts and cookies. To keep the image of the online broker user friendly and consistent on all pages I chose to use Bootstrap as a front-end framework.

The design of the application started from a basic template to slowly adding functionality and connecting it to the Rails database.

The validation was left for the end when the system was connecting and working properly with the server.

### RESTful Interoperability

The entire process is a series of Ajax requests to the different server links that run in the background just with the click of a button.

The application is using JSON objects to send the data provided by the user in the fields, these were created in jquery by constructing a form of objects that contains each of the 5 models with its attributes. This is done by setting separate class names for each of the fields that require information about separate models.

The data from the form is sent through multiple Ajax "POST " requests to the routes of the system:

*'http://localhost:3000/customers.json'*

*'http://localhost:3000/customers/'+customer\_id+'.json';*

*'http://localhost:3000/customers/'+customer\_id+'/vehicle.json';*

*'http://localhost:3000/customers/'+customer\_id+'/quote.json';*

*'http://localhost:3000/customers/'+customer\_id+'/policy.json';*

And it's retrieving through Ajax "GET"

The customer\_id needed to be firstly retrieved from the database after the creation of the customer object. This is done through another Ajax request.

### Rationale

**Cookies**

When retrieving the quote and the past information the user is sent to a second page. For the information to still be there we needed a way to remember what happened previously. In this case cookies were used to keep in the customer\_id information in order to retrieve everything on a separate page.

This means that the entire broker is using client side technology and is not dependent on another server. This is also a design strategy used for this assignment in order to store the data on a computer file but still be able to see it in a browser, thing that would have been impossible using PHP.

**The validation**

The document js\_validation.js contains the validation functions that are used on the fields. These are called when a button is clicked and information needs to be sent to the Rails underwriter.

For email, postcode and phone number I have used Regular expressions to check the form input. In case of wrong or invalid data the user would be notified via an alert box presenting the details of the issue.

**The claims**

For a more user friendly approach the claim form has been created depending on how many claims a client would want to input. This has the option to select the number of claims and depending on that the input fields would appear. This section has been realised using jquery and a loop that creates and appends html data in a specific <div> tag.

### Resultant System

The resultant system can be observed in the following screenshots and also in the screen cast provided with the Assignment Report.

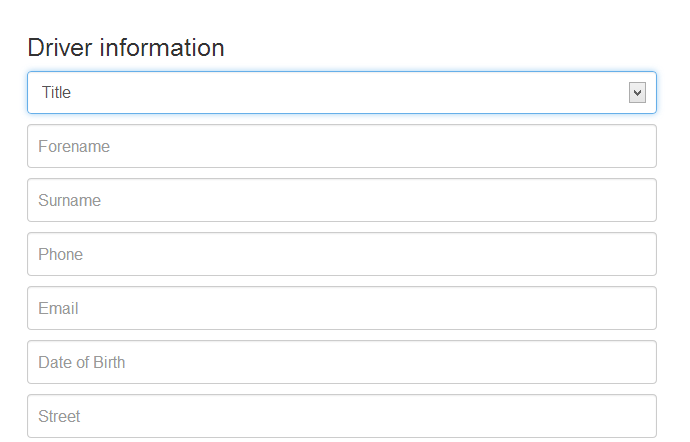


Image shows a section of the form that needs to be filled in by User.

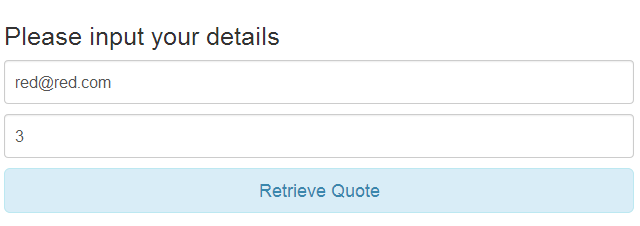


Image shows The Retrieve Quote page

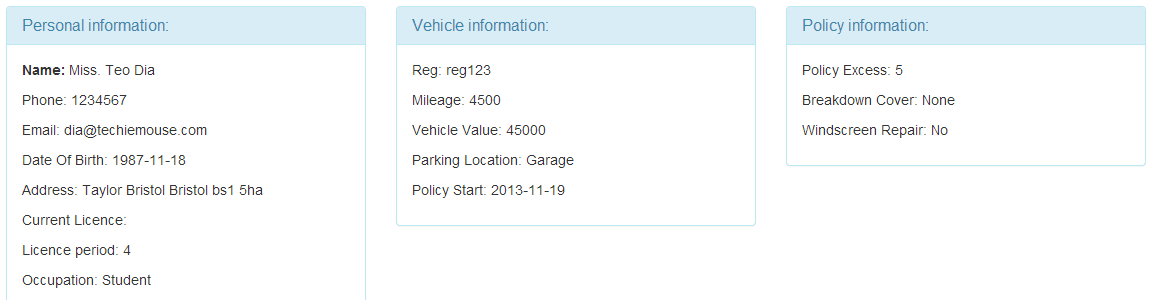
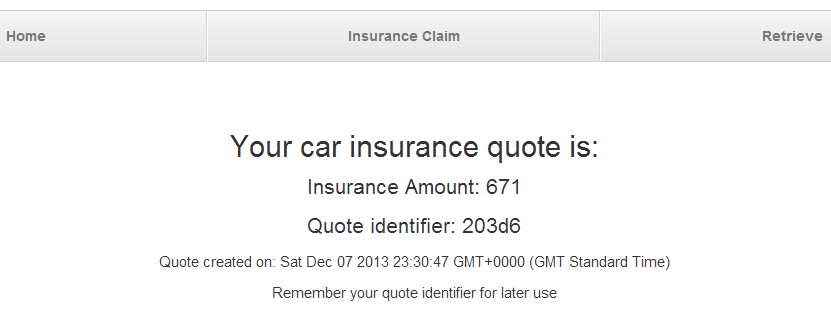


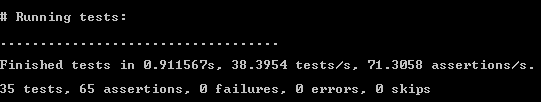
Image shows the quote page including the previous entered data.

# Testing

### Strategy

Within the Rails server the testing has been generated at the start of the system and with each migration. While the server is running the Rails tests can be started via ruby command line by using: *rake test* in the project folder.

The result of this command will provide the number of tests ran, the assertions found the failures and the errors. The Insurance application has 35 full tests and there are no errors or failures on the system created.



During coding I have also used blackbox testing and fault injections to check the error handling and issues that might occur during use.

### RESTful Client Testing

The Client side has been tested by different functional tests such as:

- Sending null data - in case a field remains empty an alert message will show up. In case of the server sending null response, the user will only get an empty response and not the word "null"

- Sending invalid data - in case the user is not providing a valid email, postcode or phone number it is announced and it can correct the problem.

- Trying to retrieve and email that is not in the database - At the beginning this was causing an error and the page wasn't redirecting. To fix this I added an error function in case the Ajax GET wasn't completed.

The Chrome Console was often use for testing and checking the Response from the server. This helped with handling validation errors, headers problems and inexistent or unnecessary JSON response.

### Results

There are 3 functional requirements requested for this system that were taken into consideration:

FR1 - Requesting a insurance quotation by inputting details into a web form.

FR2 - Saving the quotation and returning a specific identifier for it

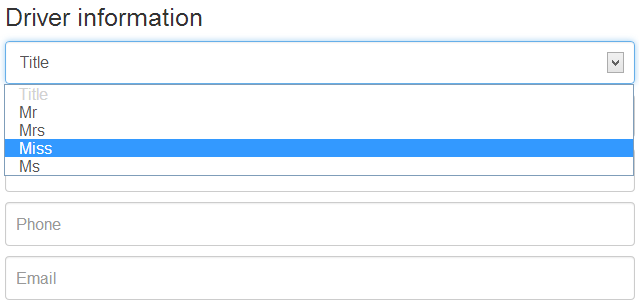
FR3 - Retrieving the same quotation at a later date

The table shows some of the functional test done on the Online Broker Application.

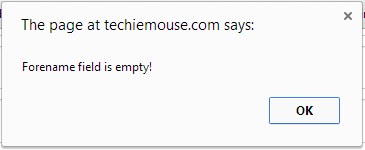
|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| ID | Requirement | Description | Input | Expected Outputs | Pass/ Fail | Comments |
|  | FR1 | Data can be provided by Customer in text fields | "Silvia" | Screen shot SS1 is  displayed | P | The form is not send if there is an empty field. User is notified of issue through alert box. |
| "" | F |
|  | FR1 | Data can be provided by Customer by selecting from list/dropdown | Choosing the Title Mr. | Screen shot SS1 is  displayed | P | The title is chosen from a list |
|  | FR1 | If field is empty data is not submitted to the database | Leaving empty the Forename | Screen shot SS2 is  displayed | P | Alert pop-up appears with message "Forename field is empty" |
|  | FR1 | Email needs to be valid | silvia@aber.com | Screen shot SS3 is  displayed | P | Email not filled in properly would result in a pop-up requesting a valid email address |
| silvia@aber | F |
|  | FR1 | Postcode is a valid UK postcode | Sy231 | Screen shot SS4 is  displayed | P | Postcode not filled in properly would result in a pop-up requesting a valid UK postcode |
| SY231PW | F |
|  | FR1 | Customer needs to input number of claims and details on each | 1 | Screen shot SS5 is  displayed | P | Selecting the number of claims would make the fields required. |
|  | FR1 | Date is easily selected from calendar | Mouse click on Date of Birth | Screen shot SS6 is  displayed | P | A calendar shows when user requires to input a date |
|  | FR1 | Windscreen repair cost is needed only if Customer requests this option | YES | Screen shot SS7 is  displayed | P | A new input field appears for the user to provide Windscreen Repair cost |
|  | FR2 | The Customer obtains an identifier for the quote he requested | Button clicked | Screen shot SS8 is  displayed | P | The page displays a 6 character identifier |
|  | FR3 | The retrieve page requests email and identifier inputs | Retrieve page accessed | Screen shot SS9 is  displayed | P | Fields for email and identifier input. |
|  | FR3 | Retrieve page redirects you to a new page that contains amount and identifier | Button clicked | Screen shot SS8 is  displayed | P | Page displays the quote information again. |
|  | FR3 | The resulted page contains the details that the customer filled in | Retrieve information button clicked | Screen shot SS10 is  displayed | P | Besides amount and identifier user can see the information provided previously |
|  | FR3 | If email or identifier are not found in database alert is displayed | Wrong identifier | Screen shot SS11 | P | If email or identifier are not found in database alert is displayed |

## Testing screenshots

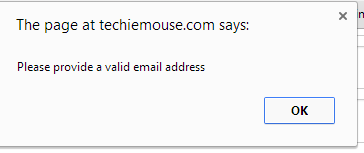
SS1



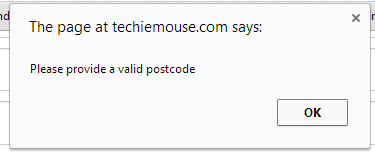
SS2



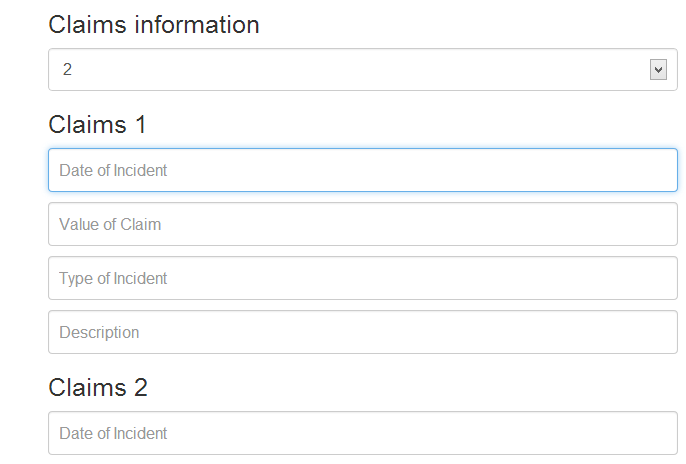
SS3



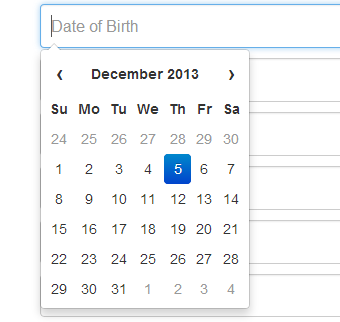
SS4



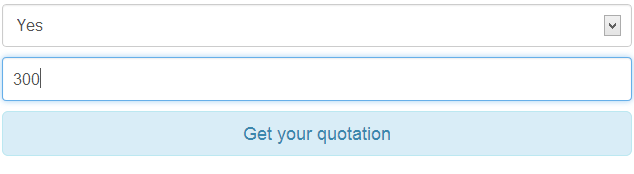
SS5



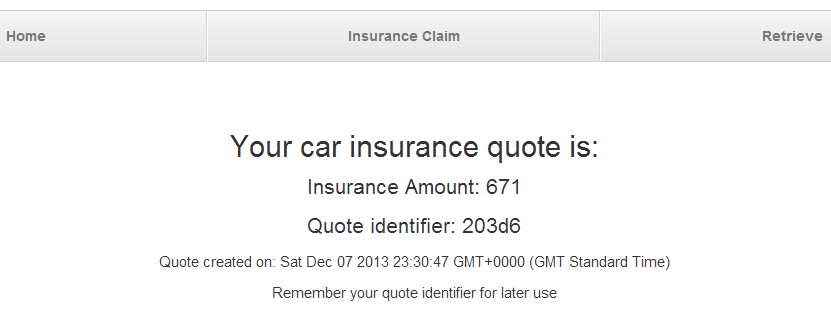
SS6



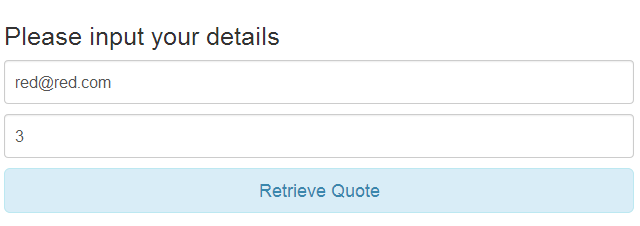
SS7



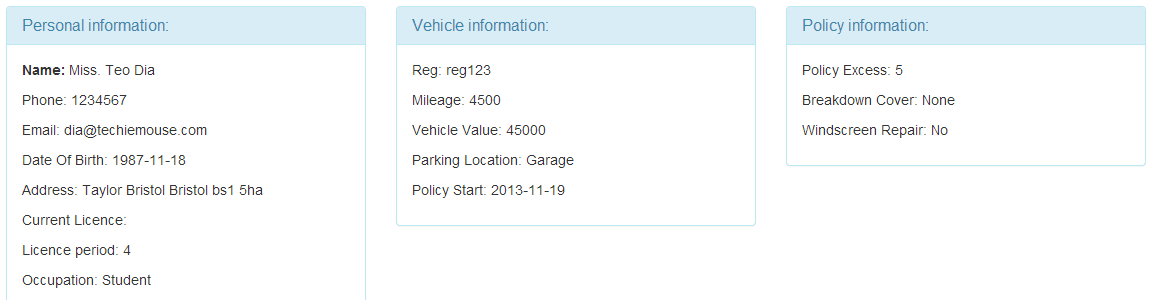
SS8



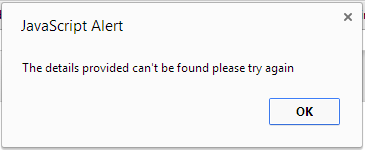
SS9



SS10



SS11



# Self Evaluation

### Approach

The project was created from the two systems: rails server and the client side. Initially after installing the Ruby and Rails with basic gems I generated the new project needed with the tables.

*rails new Online\_application*

*rails generate scaffold Customer title:string*

This was deduced by analysing the data requirements and the functional requirements.

I have used multiple migration files to add the other tables and their columns.

After the database was put in place I started thinking of the best way to code the client side and decided on Javascript and Ajax to be used for the RESTful transactions.

The next step was to connect the two systems and make sure the data is sent in the right part of the database. I used both the Rails link http://localhost:3000 (to get to the Browser View for Rails) and SQLite to check the progress of my application and the way it was working in the back end.

### Problems & Compromises

**Associations**

The issues encountered were related to making associations between the customer and all the other models. This has been resolved by changing files such as: routes.rb and all the controllers and models. References to all the models were made from customer and the relations such as "one to many" or "one to one" were set-up in the models.

The *new* function in the controller needed to change for all the models that belonged to Customer so that they were connected through a customer ID.

**Retrieving Customer ID**

Another difficulty was in sending the data for the tables that were dependent on the customer\_id. To go around this issue the first Ajax POST request was sent to create the Customer first and the complete function for this request, with the JSON response was expected before trying to send all the other information. This is due to the customer\_id being needed in the url for the other POST requests.

**HTTPS on rails server**

To use https on the rails server I have created a self signed certificate using OpenSSL and set it up by creating a secure\_rails file in the bin file of the project. This meant that the server could now be accessed through both http://localhost:3000 and http://localhost:3002 (I chose a different port in this case) and that the links in the Online Broker javascript needed to be changed to the new address.

The two issue I found with this was:

- needing a different command to start the server: *bundle exec ruby bin/secure\_rails s*

- because it's a self signed certificate this wouldn't be recognised by the browser so we need to first access the https://localhost:3002 to be able to send data from the Web App.

I wasn't able to find something to go around that and it would be an issue that would be interesting to look into further.

To implement **SSL on the broker side** an approach would have been installing a local server with apache and setting up a new self signed certificate.

**Validation -** to announce an empty field or a invalid entry I have used pop-up windows. This might cause issues where the browser has these blocked. Another option could have been using <div> attributes with specific ID's made visible only if the validation wasn't working or highlighting the fields that are required attention in case of invalid input.

### Areas of Expansion

As a future task the Car Insurance Application could be improved by adding a few more features and security options:

- Showing the number and details of claims when retrieving the quote.

- Validation rules for more of the fields: numbers, strings, date. This would increase security on both client and server side and will avoid external user to get access to data by using web attacks.

- Time option on the Policy start date: this was something that could be easily done by adding a second field in the form and using the *datetime* type on the Rails server when creating this policy parameter.

- Https on broker side - by either setting up an local apache server or by actually using a real web hosting and getting an SSL Certificate

I have also looked into Ruby on Rails validation and how the rules are added to each of the models but dues to time constraints I wasn't able to integrate them with the Client Broker correctly as when doing so the test were coming back with a few errors.

*class EmailValidator < ActiveModel::EachValidator*

*def validate\_each(record, attribute, value)*

*unless value =~ /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i*

*record.errors[attribute] << (options[:message] || "is not an email")*

*end*

*end*

*end*

*class Customer< ActiveRecord::Base*

*validates :email, presence: true, email: true*

*validates :forename, presence: true*

*end*

### 

### Output & Assessment

The Car Insurance Application passes all the functional and design requirements and also provides a user friendly front end for the user. The application is clear and robust and it uses a RESTful way of transmitting data making it scalable and portable.

Using Ruby on Rails was a new experience and I am starting to understand the Model View Controller framework. The amount of files generated for the project was intimidating and understanding what things did or what they connect to was a challenge at the begining.

Dealing with Javascript in more depth was both fun and frustrating and it made me learn more about client side, cookies and the importance of console logs. Also I believe Ajax and JSON are powerful tools that can make the connection between server and client side seamless.

I believe that my knowledge of Javascript has increased and I am now more confident with Ruby on Rails.

Suggested mark: 80% - I believe I have succeeded in creating a user friendly application that has most of the functional requirements requested. Testing has no errors and the code is easy to understand due to comments and suitable variable names.

# Acknowledgements

1. <http://edgeguides.rubyonrails.org/active_record_validations.html>
2. <http://guides.rubyonrails.org/association_basics.html>
3. <http://www.w3schools.com/js/>
4. <https://blackboard.aber.ac.uk/bbcswebdav/pid-186282-dt-content-rid-566160_1/courses/SE31520_2013-14/SE31520_Report_2012.pdf>
5. <http://en.wikipedia.org/wiki/Representational_state_transfer>
6. <http://www.w3schools.com/js/js_cookies.asp>
7. <https://www.altamiracorp.com/blog/employee-posts/configuring-webrick-to-use-ssl>