SE31520: 'MyAlcoholFreeWine.com'

Due on Monday, December 7, 2015

James Euesden - jee22

Contents

Introduction

The assignment task[1] was to implement a web site using Ruby on Rails, and a Web Service with RESTful resources. The web site, MyAlcoholFreeWines.com (MAF), was to call two instances of the Web Service as Wine Suppliers, get their list of Wines and then display to a Customer the cheapest wines available, and place an order for them if they wished. This report serves to document the design, testing and structure of my application, and as self evaluation of work carried out. All Wine images used for the application are credited to StockVault[2] from their Free Wine Stock Photos gallery.

MAF Architecture

My experience with Ruby on Rails before the assignment was very limited, and I was unsure how to begin to design the architecture. I knew I would be using MVC, and that I would have HTTP requests coming through the browser to RESTful endpoints on my controllers. The controllers would then talk to the Model classes they were associated with, and the view would display what the controller had for them.

To get a good starting point for my application, the preliminary design was based on the example provided in the 'Agile Web Development with Rails'[3] book, and then heavily modified in order to suit my own needs and the requirements of the assignment. This starting point allowed me to see the MVC structure and how to begin writing the application.

In order to fulfill the requirements, I needed a number of entities: Wines, Customers, CustomerDetails, Baskets and BasketItems. These all have their own controllers, views and models, where their data is stored in a local Sqlite3 database during developement. I also implemented a controller for Sessions, which refers to when a Customer is on the website and has a 'session', allowing them to be logged in and retain a Basket as they navigate through the site, until their session ends.

I knew that I had to have some way of storing the Wines when I got results back from the Web Services. I considered I could keep them in memory, which might be okay for a very small application, but would be completely un-scalable for an application that could be used by many people at once and with multiple massively stocked Wine Suppliers to call to. I chose to keep the Wines received in a Database and represent them as entities, where I would store all Wines provided by each supplier, but only the cheapest of each Wine.

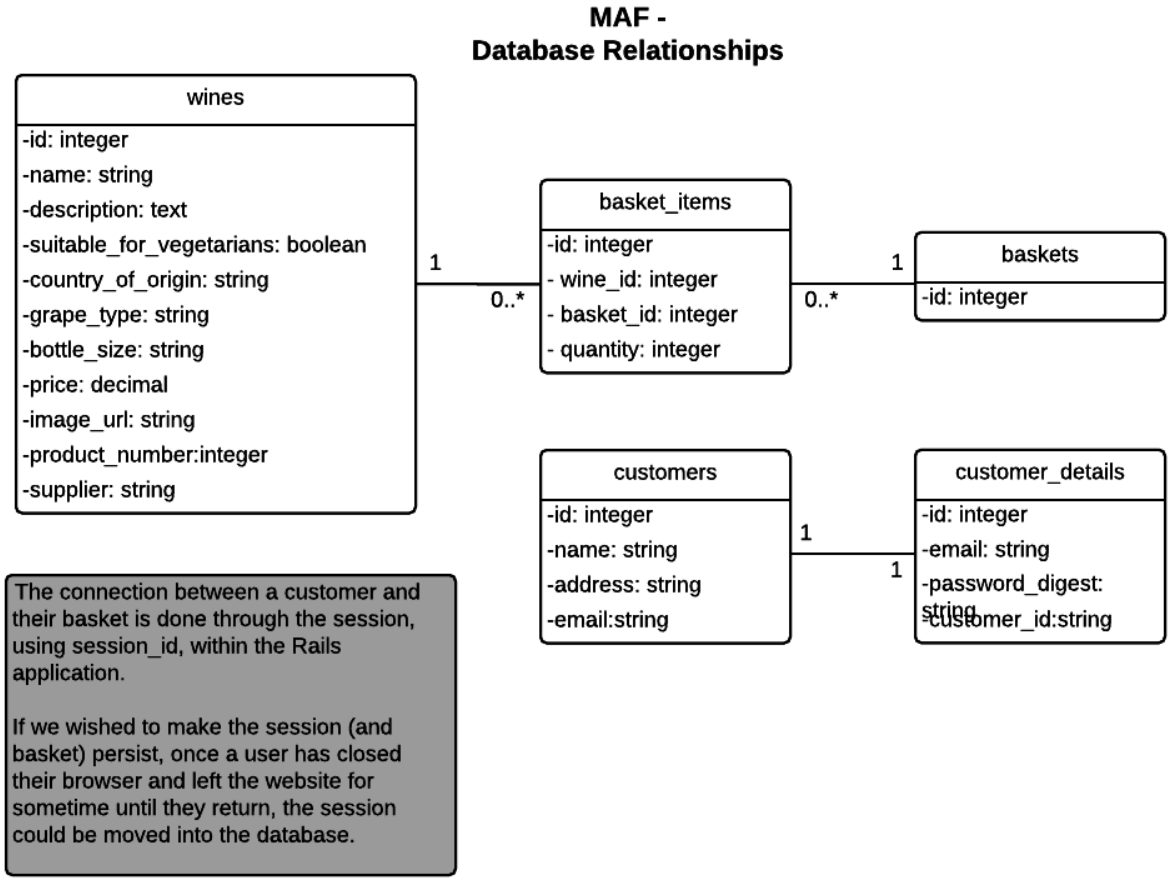


Figure 1: The intended design of the database and relations between tables in the MAF application.

The representation of a BasketItem is for a many-to-many relationship, indicating that a Basket can have many Wines in it, and an Wine can be in many Baskets of different Customers. The connection between Customers and their Basket is done through the session parameter in Rails. I did consider having Session as an entity for storing in the database, and this would allow a user to retain their Basket and BasketItems for another time when they next login. However, I felt this was out of scope for the prototype and decided it was okay to have the basket be emptied and removed when a Session is ended. This leaves the session parameter containing the Basket id for a customer, and if they are logged in also their id and email, to make the join between the two.

I chose to keep the Supplier name as a field in the Wine entity, and the connections to these Suppliers is located in a configuration file. This could allow Suppliers to be updated through configuration once the site is hosted. However, I also know that there is the possibility of having a Table in the database for Web Suppliers which could contain their addresses, names and any additional needed information. Were I to re-implement this application, I may opt for that approach should more information about the Suppliers be required.

Based on what I expected to need, I made the target class design diagram that can be seen below. This shows the connections between the requests to MAF controllers and what the controllers connect to for their data and showing it.

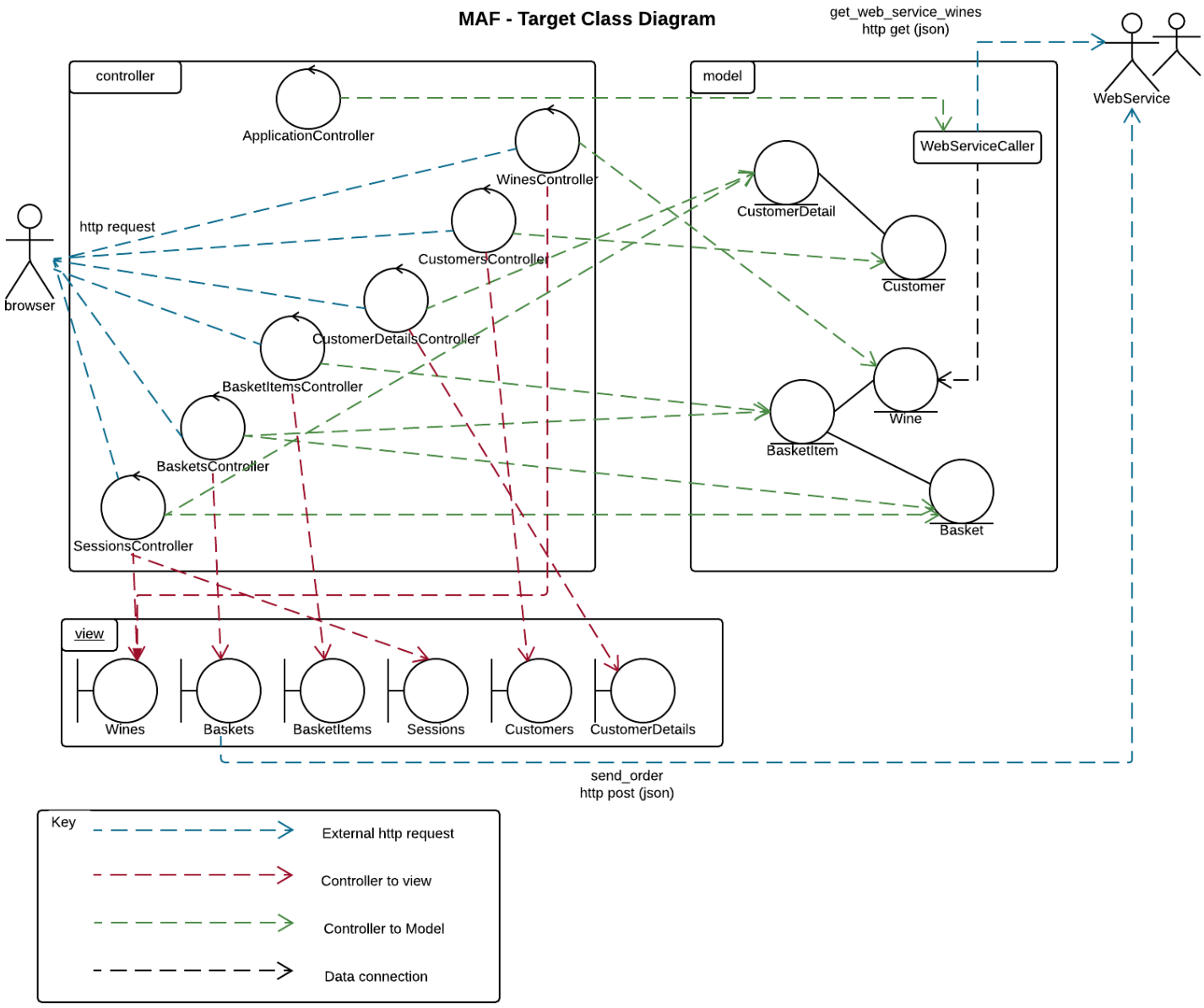


Figure 2: The intended design of the classes and their connections in the MVC pattern.

You can see that the SessionsController interacts with the Basket and CustomerDetail model, in order to keep the customer logged in with a persistent basket while they browse.

To get the Wines, HTTP POST requests are scheduled to be made every 60 seconds to each known Web Service, using rufus-scheduler gem[4] and are made asynchronously with SuckerPunch gem[5], from my own class WebServiceCaller. This ensures that a user is not

interrupted while they browse and the Wines are always kept up to date without making calls to the Web Suppliers too frequently.

The WebServiceCaller gets the list of Wines from the Web Services listed in a config file, checks to see if the Wine is cheaper from a different supplier than the one known and updates the Wine to reflect the alternate supplier's wine if it is cheaper. Otherwise it just confirms there are updates to the Wine given and updates it if there are. Wines between different suppliers are distinguished by their product name only. In a real version of this application, a unique universal identifier would perhaps be more appropriate, such as a bar code.

For the images of the wine bottles, I chose only to get the image url from the Web Service that matches with images already in MAF, rather than to link back to the Web Service. Through doing this, it leaves the application open to be able to be expanded upon later to get an actual copy of the images and store them in assets. I feel this is a better choice than linking back to the original images, as the bandwidth usage would increase on the Supplier web site through our image link back to them.

In order to fulfill the functional requirement of Search, I decided to use Solr with the Sunspot gem [6]. This allows for fulltext search and partial text search using NGrams, through any fields set as 'searchable' on an entity. The allowing of partial search needs to be set up in the scheme configuration of Solr in order to work, but is very powerful once it is running. In order to use Solr, we must set up a local Solr server to use, and allow it to reindex for our records in order to find them.

```
1 bundle exec rake sunspot:solr:start # or sunspot:solr:run
2 rake sunspot:reindex
```

There are a number of gems to be used for Search, or even writing a method to do it for us through SQL queries. I chose to use Solr, however, as it had exactly what I needed in order to search through my list of Wines, without me trying to re-implement exactly what it does already.

I chose to keep Customer and CustomerDetails separate, where CustomerDetails actually refers to an id, customer email and their password digest. By keeping the Customer and CustomerDetails separate, it allows us in future to allow Customers to update their information in the Customer table without affecting or accessing the CustomerDetails. Similarly, if an attacker found their way into the Customer table, they wouldn't automatically also have access to the CustomerDetails table too, giving us added security through this separation.

Sending Orders to the Web Service is as simple as when a Customer clicks Checkout (while logged in), the order is sent to a method where a JSON body is built into a POST request for each Wine, and the order(s) is then sent to the respective suppliers of the Wines requested by the order. In a future version of the application, it would be preferable to have Orders kept in the database, to allow a Customer to see their previous orders.

Web Service Architecture

Since I was new to Rails, I chose to start developing my Web Service first, using Rails over any other language to get comfortable with Ruby on Rails. The Web Service only needed to know about two things, Wines and Orders, with RESTful routes to access what is needed for them, in this case a GET for wines.json, and POST for orders, to place an order and store it in the database. With this in mind, my target classes before implementation are shown below.

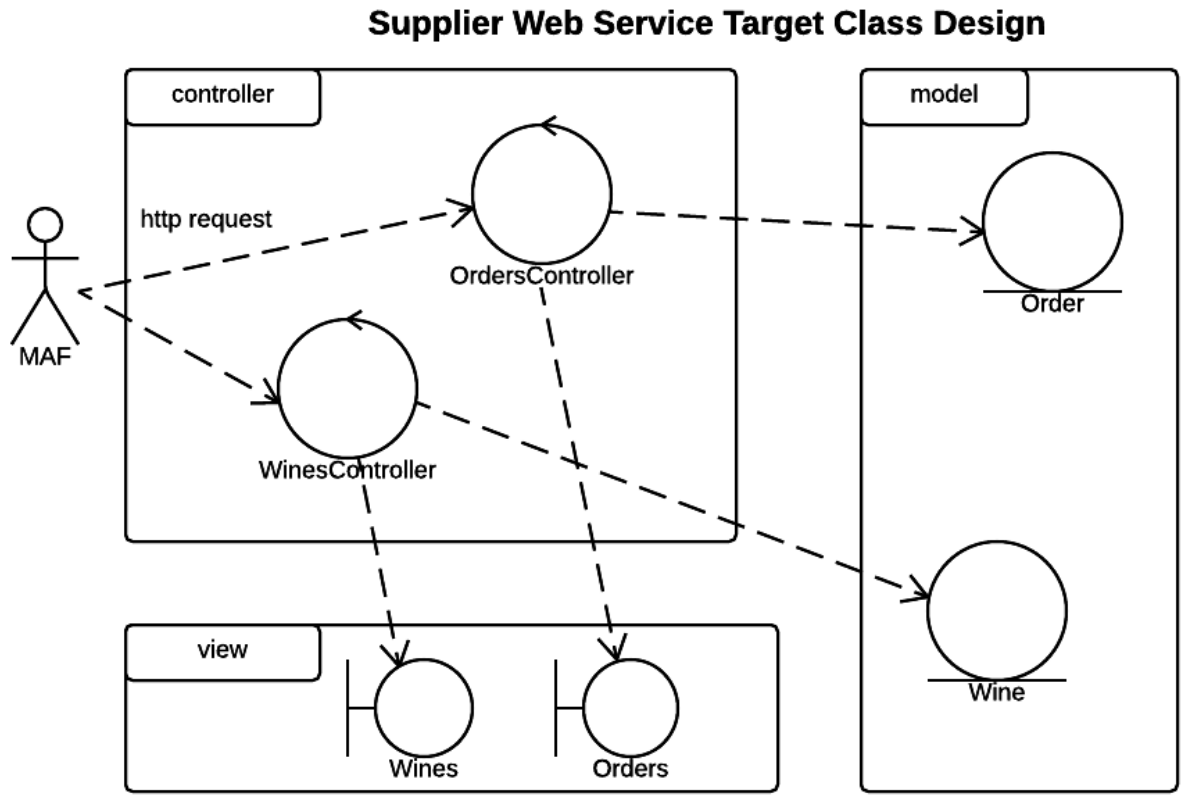


Figure 3: The intended design of the classes and their connections in the MVC pattern.

Using RESTful endpoints, we don't need to define specific routes for certain things, and can just let Rails handle it, as long as we define that Orders and Wines are resources. When providing the Wine data, a GET request comes from MAF to the WinesController that includes a custom HTTP header for when the service was last accessed by the MAF. The Web Service uses the time value from this header and gets all Wines in the database that were created or updated after the time MAF last asked for them and returns them in JSON format, along with its name, e.g.

```
1 {"id":14,"name":"Abracadabwine","description":"Better than Magic! Tasty!",
    "image_url":"wine6.jpg","price":"5.01","country_of_origin":"Poland","
    grape_type":"Green","suitable_for_vegetarians":true,"bottle_size":"350
```

```
ml","url":"http://localhost:3001/wines/14.json","supplier":"Wine
Supplier A"}
```

Doing this cuts down on the amount of database operations on the MAF side, and the bandwidth used by the Supplier to return the Wines. I originally had a method for deleting Wines no longer existing in any Web Service which worked until I implemented the check to only send wines that had been created or updated since I last called it. The delete function then started deleting all Wines in MAF except those updated.

I commented out the function, as it is no longer able to be used with the way I have implemented the communication between the MAF and Web Services, but left it in the code base for potential future reuse. However, if I were to do this again, I would have a flag on each Wine in the JSON sent by the Web Service that says what status it is in (In Stock, Out of Stock, Discontinued) and use this flag to determine whether to keep or delete a Wine from the MAF database.

When it comes to receiving Orders, the Web Service OrdersController receives a body of JSON that it parses into an Order entity and puts into the database if it passes the model validation, which is enough for this prototype. Each order contains the required customer information (name, email, address), the ID of the Wine in the Web Supplier (saved as 'Product Number' in MAF) and the quantity of the Wine to be ordered. I included a view for the Orders so I could see that the Orders were correctly sent by MAF.

Test Strategy

System Testing

I helped myself focus on implementing the functional requirements through using a system test table, indicating what the system should be like and how it should respond based on assumptions the functional requirements. I wrote my system test table after reading the requirements of the application and gradually went through them one by one as I implemented my design, to ensure my system met the requirements.

The majority of my system tests passed, and completing each functional requirement and fulfilling each system test was pretty satisfying. If this system were more than a prototype, I'd advance on these system tests much further to include more in-depth testing on system validation of user input, the security of sessions and more examples of searching for Wines using Solr. The system test table and my results from it can be seen below.

MAF System Test Table

ID	Requirement	Description	Inputs	Expected Outputs	Pass/Fail	Comments
A1.1	FR1	Customer can browse all cheapest wines in paginated (alphabetically ordered) list	n/a	All cheapest wines from the suppliers are displayed in alphabetical order, and are paginated.	p	
A1.2	FR1	Customer can browse all cheapest wines in paginated (alphabetically ordered) list: When a wine is updated, we see this reflected in the wines list	n/a	The wines list is updated to relfect the change of wine(s) in the supplier, e.g. the description of a wine has been updated	p	
A1.3	FR1	Customer can browse all cheapest wines in paginated (alphabetically ordered) list: When a wine a cheaper wine by a different supplier is found, display this wine in the wine list as the cheapest, no longer showing the previous wine by the other supplier.	n/a	The wines list is updated to show the new cheapest wine by the alternate supplier	p	
A2.1	FR2	A customer can Search for any Wine by full word, and any Wine that has any information that matches that search will be returned.	The search term request	The Wines that match the search term request, be it from name, description, country of origin, etc, will be shown	p	
A2.2	FR2	A customer can Search for any Wine by partial word, and any Wine that has any information that matches that search will be returned.	The search term request	Any Wines that match the partial term will be returned and shown to the customer.	p	
A2.2	FR2	A customer Searches for text that is not associated with any wine	The search term request	No Wines will be returned, and the user is informed that their search returned no results	p	
A3.1	FR3	Display detail of a Wine when selected by a User	The wine to be viewed	The full information about a wine will be shown Name, Description, Price, Supplier, Country of Origin, Bottle Size, an image of the bottle, Grape Type and if it is Suitable for Vegetarians.	p	
A4.1	FR4	A Customer can add a Wine to their Basket from the Wine view	The wine to be added to the Basket	The Wine they selected to be added to the Basket is added to their Basket	p	
A4.2	FR4	A Customer can add a Wine to their Basket from the full Wines list view	The wine to be added to the Basket	The Wine they selected to be added to the Basket is added to their Basket	p	
A4.3	FR4	A Customer can specify the quantity of the Wine they wish to purchase and then add that amount of the Wine to their Basket	The wine to be added to the Basket and the quantity of the Wine	The Wine they selected to be added to the Basket is added to their Basket, in the quantity they selected to add to basket	f	Quantity of Wine to add to Basket not implemented for this prototype, only one of the Wine selected may be added to the basket on the button click, but a Customer may have as many of that Wine as they wish in the Basket.
A5.1	FR5	Display Shopping Basket Shopping basket empty	n/a	Do not display an empty basket	p	
A5.2	FR5	Display Shopping Basket Wines displayed in shopping basket that the user has added to their basket, with the quantity.	Basket_items that the customer has placed into their Basket.	Display the Basket with the customers selected wines and their quantities	p	
A5.3	FR5	Display Shopping Basket Remove a wine from the shopping basket.	Items in the basket to be removed.	Display the Basket with the customers selected wines and their quantities, and remove an item when 'Remove' is clicked beside a Basket Item. A notification will inform the user that their Basket Item has been removed.	p	
A5.4	FR5	Display Shopping Basket Empty the entire shopping basket.	Items in the basket to be removed.	All items are removed from the Basket and no Basket will be displayed. A notification will inform the user that their Basket has been emptied.	p	

MAF System Test Table

A5.6	FR5	Display Shopping Basket Change the quantity of the wines in the shopping basket	Items in the basket to have quantity changed, and quantity differences.	A Basket Item can have its quantity changed	f	Out of scope for prototype requirements.
A6.1.1	FR6a	A logged-in Customer may proceed to Checkout and place an order for items in their Basket	The customer is logged in and has Wines in their Basket they wish to purchase	The order is sent to the Wine Suppliers of the Wines the user wishes to purchase	p	
A6.2.1	FR6b	A Customer who is not logged in is prompted to Login before they can Checkout and purchase the Wines in their Basket	The Wines in the Basket the Customer wishes to purchase	The customer is prompted to login when they click to Checkout, and once logged in are returned to their Basket view to try Checkout again.	p	
A7.1	FR7	A Customer who is not logged in can use their email and password to login to their account	The customers email and password	The Customer is logged in and given a notification that they are now logged in. The user who is currently logged in is displayed in the header of the website	p	
A7.2	FR7	A Customer who is not logged in attempts to login with bad credentials (unknown email/password)	The customers email and password (invalid)	The Customer is informed that their email/password was invalid and is not logged in	p	
A7.3	FR7	A logged in Customer may click logout to exit from their account (and also remove their session/basket)	The session must be aware of the current customers basket and id	The Customer is logged out of their account and informed that they have been logged out	p	Note: In the current implementation, the Basket is not persistent once a user logs out. In a future version, once the Customer is logged out it is still possible to retain their Basket for their next login visit. Out of scope for prototype.
A7.4	FR7	A Customer who is not logged in is able to make a new account The account being created is using an email address not yet known to the system	The Customer details, email and password	The Customer account is created, they are told it has been made and are requested to login	p	
A7.5	FR7	A Customer who is not logged in is able to make a new account The account being created is using an email address already known to the system	The Customer details, email and password	The Customer is informed that the email address is already in use and they must supply a different email address	p	
A7.6	FR7	A Customer who is not logged in attempts to make a new account The Customer fails to supply some of their details or email	The Customer details, email and password, with some information left blank	The Customer is informed that they have not supplied some required information, and asked to try again.	p	
A7.7	FR7	A Customer who is not logged in attempts to make a new account The supplied confirmation password does not match the given password	The Customer details, email and password, with wrong confirmation password	The Customer is informed that their password and confirmation password do not match and they must try again	p	

In order to back up my system test table, along with the screen cast provided with the application and this report, I have included some screenshots of the application in various stages of the system tests. Not all tests have been included in the screenshots, and some tests are difficult to give examples for through screenshots alone.

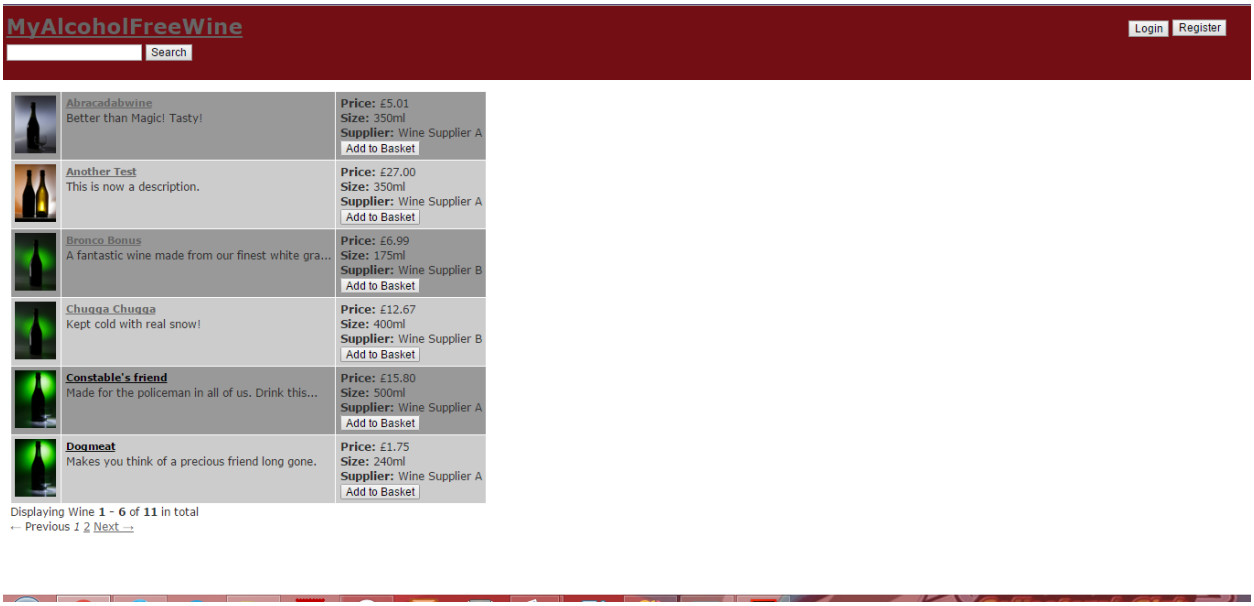
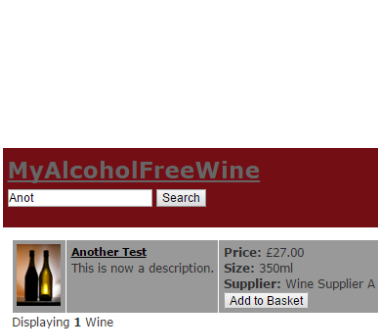
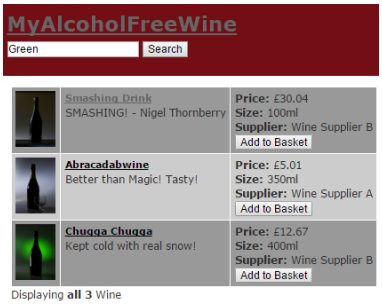


Figure 4: FR1 - Displaying a list of cheapest Wines.

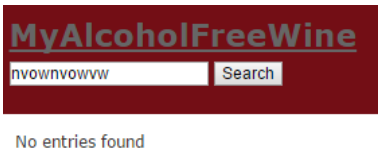
The system presents the user with a number of Wines, provided by the Web Services. If a Wine in a Web Service updates, we can see the change reflected here. Through testing this requirement I was able to see when the bug of all wines except those recently updated were deleted appeared and fix it.



(a) Search using partial text to find a Wine



(b) Searching Green for Grape Type

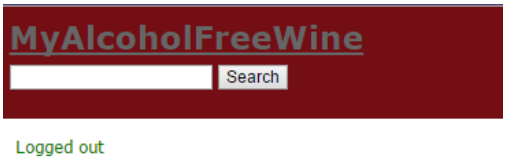


(c) Searching something that won't give results

Figure 5: FR2: Using the Search bar

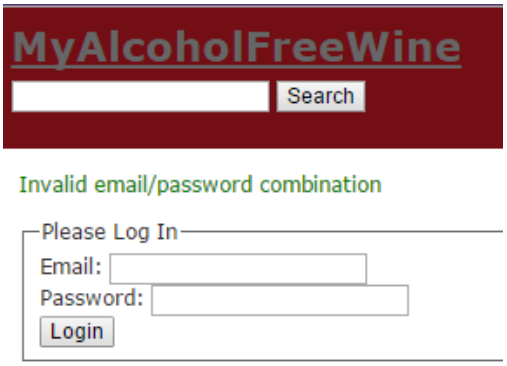
Figure 6: FR3 - Displaying the full info for a Wine, FR4 allowing the Wine to be added to the Basket.

(a) The login form



(b) Informing the Customer that they have Logged out

Figure 7: FR7: Customer account actions



(a) With invalid email/password

(b) Successfully creating a customer account

Figure 8: FR7: Customer account actions

There are a number more system tests that could be done, and if this were a full project I would have system tests for every route that could be accessed, add security to the customer sessions, potentially use Selenium for automating tests of browsing the web site as a user to see what happened under certain conditions and more. As the majority of the conditions laid out in my functional requirements systems test table pass, I am pleased with these results. However, if given more time I would like to look into other ways of testing the system to be sure that it was as foolproof as possible.

Unit Test for models and controllers

For testing the innards of my application, I used Rails unit testing for my models and controllers. Some of the generated tests by Rails were no longer required, as I didn't need actions for things such as New for Wines, as they are added into the database by the WebServiceCaller and not required to be added manually in MAF.

The majority of my Controller unit tests are ensuring that new objects are created when requesting, and importantly that the user is redirected to where they need to go. An example is when a User creates a basket item (adding an item to a Basket). I don't want a user to go to the Basket items list and see a bunch of items from all Baskets, and so they are instead redirected to their particular Basket_path.

My unit tests for the models are mainly checking the validation, that no entities will be created that violate the conditions of their existence or creation. This ensures things a Customer always having an email address, and that it is unique to itself.

```
D:\Users\James\University Documents\Modules\SE315\Assignment\SE31520-MyAlcoholFr
eeWine\MyAlcoholFreeWine>rake test
DL is deprecated, please use Fiddle
Run options: --seed 59521

# Running:

.............................

Finished in 1.221823s, 25.3719 runs/s, 41.7409 assertions/s.

31 runs, 51 assertions, 0 failures, 0 errors, 0 skips
```

Figure 9: Asking rake to run all test for my MAF project.

Self-Evaluation

Summary

Overall, I'm relatively happy with the result of this assignment. While I don't have full confidence that everything I have done and my design decisions are correct, I feel like I've learned a large amount. Before the assignment I wasn't comfortable writing a Ruby on Rails app at all, and not too sure on routing and communication between web applications either. Now I feel happy enough that I could attempt to write more Rails apps and have enough experience to get going by myself, but still have much more to learn.

I know there are areas for improvement (security, clean up, persistent sessions and baskets, better tests, cucumber tests, etc), but as far as the time allowed, my understanding and learning of Rails and this assignment tasking me to just create a prototype, I am happy with the resulting program. I appreciated the help and starting point of the Agile Web Development with Rails[3] book, and think if I were to re-implement this application with better knowledge, I would still use many ideas from this book to begin with and build upon them just as I have done for this.

During implementation, I found the most difficult part to be implementing routing, and getting my links in the views to execute the correct controller actions as I wanted (for the more complex tasks, such as removing the last item from a basket, logging in and sending the webservice order). Before this assignment I had little working knowledge of rails outside of the workshops and a small amount of experience with routing, and doing this assignment helped me get my head around routing and how it works. With little knowledge of Rails, it was difficult to get started, but once I got into the flow of development and started to understand more how Ruby on Rails works, it became much easier and more enjoyable.

Mark Breakdown

Screen cast

My screen cast shows my MAF fulfilling the functional requirements set out by the assignment, and two Web Services running with separate data and MAF communicating with them.
Mark: 9/10%

Design

My sections for the architecture of MAF and the Web Service both display diagrams for my classes (MVC) and there is a diagram to show the relationships between my tables in the database for MAF. I have described the way in which my application works and why I chose certain ways of implementation. My Rails knowledge is not fantastic, but I did the best I could with what I know to design a system in the 'Rails way' and use MVC appropriately to design a system that would fulfill the requirements.
Mark: 17/20%

Implementation: MAF

My MAF application runs, using Rails the architecture for MVC and communicates with the Web Services. The code is commented where appropriate, identifier names make sense for what they represent and adheres to DRY. Data manipulation is handled in the Models, Views stray away from logic code and Controllers are thin, getting data from model methods and enabling the views to display it as necessary.
Mark: 23/25%

Implementation: Web Service

One Web Service application was built, with two instances of it able to be run, with different data sets. The resources for the Web Service are RESTful and requested are correctly routed to GET and POST requests where appropriate (wines, orders). The Web Service only returns those items that have been updated since the MAF last called to improve efficiency of sending data.
Mark: 15/15%

Testing

There are unit tests for each controller, and more than just the ones generated by rails, and unit tests for the models where appropriate for the validations and logic methods. There is a system test table that matches my assumed customer expectations. I have discussed the results and how I used the system test table to work through the functional requirements.

There are, however, no cucumber tests and testing could cover more areas.
Mark: 10/15%

Evaluation

I have indicated the results I believe I would receive for each section defined by the marking grid and why. I have a summary of my evaluation to discuss what I found challenging or easy about the assignment, and what I learned through doing it.
Mark: 5/5%

Flair

Some flair through my implementation of Search using Solr with Sunspot to search partial words from all sections, editing the schema to change the gram of the search, calling the WebService asynchronously with SuckerPunch, scheduling it to update with rufus-scheduler and only retrieving the latest updates to wines to cut costs on database operations. Could be more, i.e. deployment to Heroku, better CSS design with Bootstrap and more user controls.
Mark: 5/10%

Total

Mark: 84/100%

References

[1] Chris Loftus, "MyAlcoholFreeWine.com",SE31520/CHM5820 Assignment 2015-16, October 27 2015

[2] Stockvault.net, 'Stockvault.net - Free wine Stock Photos', 2015. [Online]. Available: http://www.stockvault.net/search/?query=wine. [Accessed: 06- Dec- 2015].

[3] D. Thomas, D. Hansson and L. Breedt, Agile web development with rails. Raleigh, N.C.: Pragmatic Bookshelf, 2005.

[4] GitHub, 'jmettraux/rufus-scheduler', 2015. [Online]. Available: https://github.com/jmettraux/rufus-scheduler. [Accessed: 06- Dec- 2015].

[5] GitHub, 'brandonhilkert/sucker_punch', 2015. [Online]. Available: https://github.com/brandonhilkert/sucker_punch. [Accessed: 06- Dec- 2015].

[6] Sunspot.github.io, 'Sunspot: Solr-powered search for Ruby objects', 2015. [Online]. Available: http://sunspot.github.io/. [Accessed: 06- Dec- 2015].