SE31520/CHM5820: Assignment 2015-16
MyAlcoholFreeWine.com

Evdzhan Mustafa (enm3@aber.ac.uk)

December 7, 2015

Abstract

Report for the SE31520 assignment during the 2015/2016 academic year, semester one.

1 Introduction

This document explains the solution to the MAF assignment. It includes section on the MAF architecture, the Web service architecture, the undertaken test strategy, a self-evaluation section, and finally environment section.

2 MAF architecture

My approach to design the MAF application was to read the requirements spec, and write down any entities, that have common attributes and actions. Doing that I came up with the following entities:

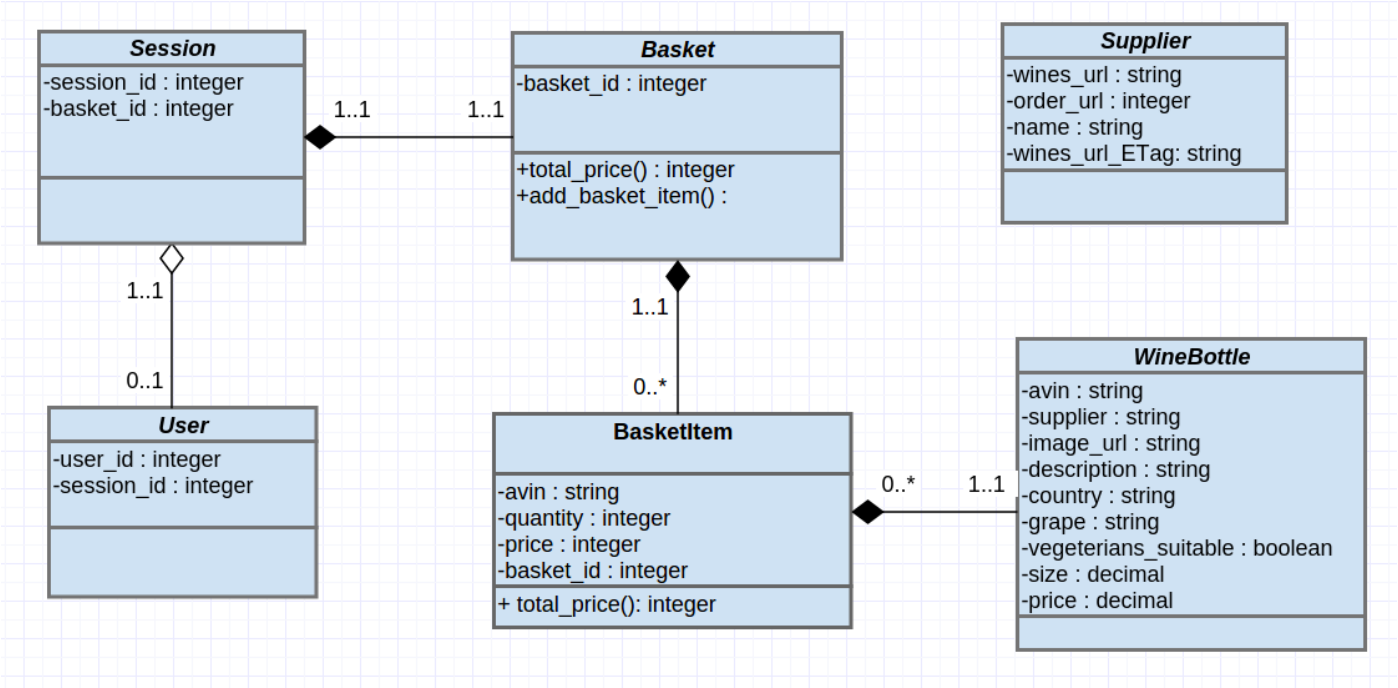
- User – has the ability to view the MAF website front end. The MAF application needs to differentiate between different users. To do that, my MAF implementation has User model. The User has the following attributes :
  - email
  - password
  - name
  - address
- Session – a placeholder using which we keep track of individual visitors of our MAF website. Has only one attribute, which is the session_id.
- Basket – entity that holds the desired wines for each visitor of the MAF website. A basket is not bound to the User entity – rather, it is bound to the session entity. A basket has no specific attributes that define it. Rather, it holds reference to the currently live session (session_id), and any items added to the current basket.
- BasketItem – a link between a basket and a wine. Basket item has the job of holding reference to a single wine and a single basket.
- WineBottle – entity representing a wine sold by a supplier. Along with the required attributes described in the spec, I have added the AVIN attribute. AVIN is basically what people have come up with when trying to uniquely identify wines. It is what ISBN is for books. Here you can find out more about AVIN. All wine bottle attributes :
  - supplier
  - image_url
  - description
  - country
  - grape
  - vegetarians_suitable
  - size
  - price
  - avin
- Supplier – this is the entity that was not explicitly visible by reading the spec. A supplier holds data about each supplier. The data consists of link to the resource that the supplier has exposed to view the wines, link to send orders to, name of the supplier, and a ETag string, that is used for to cache the wines data in the MAF application. Here are the attributes :

- wines_uri
- order_uri
- name
- resource_etag

2.1 UML Diagram

A high level view of my MAF implementation can be seen in the following figure :

Figure 1: MAF design as UML



Session – has one, and only one basket associated with it at any given time. Session has optionally a single User associated with it. The decision for this was made so that, non registered visitors of the website can also have a basket. Session is provided by Rails by default and I do not save it my self anywhere. This covers Functional Requirements FR4, FR5 and FR6b.

User – has a single session, which changes every time he logs in and out. This makes the basket non-persistent. That is, if a existing User, adds wine to his basket, logs out, and then logs in back again, the wine in the original basket will be gone. No explicit decision was made over this. The requirement spec does not state how this should be done. This could be easily changed, but focus was put on other areas in my MAF implementation. User is implemented using the devise Rails gem, and is stored on the MAF database. The User covers Functional Requirements FR6a and FR7.

Basket – belongs to a single session. It has 1 to 1 relation with a basket. Neither can exist without the other. Basket has zero or more basket items – zero when the user has first entered the website, or emptied his basket, and many if he has added items to it. Basket is saved to the MAF database. A basket covers Functional Requirements FR4 and FR5.

BasketItem – belongs to a single Basket. BasketItem cannot exist without basket associated with it. BasketItem has one and only one WineBottle. Basket item has avin, quantity and price fields. The avin is to link it with a WineBottle. The price is recorded for each BasketItem in case the Web Service provider, changes price of any of its wines. As cached wines were introduced to my application, recording the price at the moment when the user/visitors adds a wine bottle to his basket, can be used to notify the user if a price for a wine in his basket has changed. I did not implement this system, as I had no time to do so. Basket is saved to the MAF database. BasketItem helps satisfy Functional Requirements FR4, FR5, FR6a and FR6b.

WineBottle – belongs to zero or many BasketItems. That is multiple users can have the same WineBottle in their basket, or no user has it in his/her basket. WineBottle is uniquely identified by the avin field, which I previously discussed. WineBottles are not saved on the MAF database. This decision was made because I didn't want to duplicate essentially the same database in the supplier Web Service application. Rather than having it into the database, I have configured my Rails application to initialize an array, that has the cheapest wines among all suppliers. Each time user/visitor requests to see the wines, the wines are refreshed from the suppliers (if needed), and are shown to the viewer. This was done using

ETags, and reading response headers from the Web Service. There is also a background thread that checks every 60 seconds, if wines need refresh. The re-check period can/should be parameterized, but I couldn't figure out how. WineBottle lives in config/initializers/wines_from_suppliers.rb. WineBottle implements functional requirements FR1, FR2 and FR3.

Supplier – implemented using simple Ruby class, and holds details about suppliers. Supplier lives in config/initializers/supplier.r
Suppliers are used by wines_from_suppliers.rb. Suppliers are not saved to the database.

2.2 Implementing the design with Rails

Initially, my design mapped in Rails into 4 controllers; later that number went down to 3. The handed in version has StoreController, BasketsController and BasketItemsController.

Most of the action happens in StoreController. The root of my MAF website is served by StoreController's index action. It gets all wines, sorts, paginates them, and serves them as html to the user. From there on any other action plays around those wines. The search action, reads a query parameter, passed in when the user clicks the search button, and looks for a matching wines in the wines array. It then displays them. Error checking is added to see the query string is empty, or if no wines actually match the query. The user is notified if the latter is true, while the former just displays all wines. The StoreController also renders a single wine page, via its show action. The rendered page shows all wine details. The last action of the Store controller is the check_out action, which holds the logic of when a visitor can checkout. If he isn't logged for example, check_out will redirect the visitor to the log in page.

BasketsController has only one method defined, which is destroy. The destroy action is invoked when a user empties a basket, or when logs out. Since a basket is bound to a session, we make sure the corresponding session forgets about the basket id, thus we effectively destroying the session too. An attempt to make this function faster was to put the destroying in to a separate thread, as I did observe strange delays while emptying my basket, I did notice improvement with a separate thread, but it might have been a placebo – I cannot know for sure.

BasketItemsController also has only one action – the create action. This action is executed when a user/visitor adds item to his basket. This happens both both via AJAX and regular post request. I have tested that using browser with java script disabled, and it seemed to work. Implementing the basket item addition via AJAX was a bit clumsy, but I managed to do it. I am responding to AJAX requests, with neat JavaScript, that highlights the contents of the user's basket. The idea for that I got from the book "Agile Web Development with Rails 4".

The final fourth controller I had was WineBottlesController, which got obsolete, as my design changed. My design changed several times during my implementation phase. Initially, I had the Wines saved to the MAF database, but did see huge flaw – I had the very same Database that backs the supplier web service. I did "DRY" the duplication out, and changed my design so that the wines are not a database table. The wines were moved to be in the memory, and continually refreshed by a background thread.

I can't say my design is perfect, after all it was my first big Rails project. There are many things to improve on, I have left myself TODO comments to remind me what could be improved. What I like about my design is that I am not duplicating the wines, as it is on the Web Service data base. That is I am storing all the wine details in the memory, instead of a database. This should make wine access faster for the user. There is a caveat at my approach however. How scalable is this approach? How many wines would a real site fetch from real suppliers? Those questions I would seek the answer to, if it was building a real application.

3 RESTFul Web Services

My Web Services were implemented using Rails. I have actually implemented only one Web Service, that uses two different databases, each populated using rake db:seed, with different price for a bottle with same avin. I have provided steps to run my web service twice in the file configure_suppliers.txt. To talk to the API I have used the HTTParty gem.

Exposing the Wine Bottles resource was quite easy. All I had to do is create scaffold with model named WineBottle, and fill it with some data. Then the said model was available when the Rails application is run my accessing localhost:3000/wine_bottles.json link. There were some configurations I had to do in order to turn the application to API. For example I had to disable the Cross-site scripting protection in app/controllers/application_controller.rb. I did create separate controller, which has single action that accepts incoming post requests, which are expected to be orders. No validation is done whatsoever, but I could imagine what I need to do to treat the incoming data and validate it. The web service is basically two Rails controllers, one expecting orders, while the other, providing json data about the wines.

A detail worth mentioning is that, I had to configure the index action of the WineBottles to return a response with code "304 Not Modified" if a content "ETag" was included the incoming request, and the content didn't change. What this means is, if the resources held by the Web Service API, doesn't change between to get requests, and a consumer gives

us a ETag that was equal to previously returned ETag, we simply respond with 304. Such response carries no body, and is only a header. This makes the API consumption more efficient, as it saves time on both sides of the communication. Our MAF web app consumer, reuses the old, but still valid wines. And the serving API, saves time by simply returning a header, instead of potentially going through the entire database, to fetch all the wines, and send them in a response body.

4 Test strategy

I have used Rails controller/model tests to verify correctness of the code, as I was writing it. I did not use any cucumber testing. I did do a lot of manual testing of my Web Application, always referring to the requirements spec to see how far I have gone. I do believe I do cover all functional requirements fully. A minor mistake I did is not doing TDD using cucumber tests. As I was new to the entire Rails framework, I felt that the overhead of writing those tests would be greater than the benefit. As a student who has been on a IY, and has done extensive testing, I know how important testing is. If I could re-do the project, I would first set out in stone the tests that the application must pass to, before I do any coding. But as I was unfamiliar with the technology, I rushed in to learning it, rather than blackbox-test it! Here is my test table:

4.1 FR1

ID	FR	Description	Inputs	Expected outputs	P/F	Comments
FR1.1	FR1	go to MAF.com	-	Wines are paginated	Pass	-
FR1.2	FR1	go to MAF.com	-	Wines are ordered alphabetically	Pass	-
FR1.4a	FR1	go to MAF.com	-	Wines have picture	Pass	-
FR1.4b	FR1	go to MAF.com	-	Wines have description	Pass	-
FR1.4c	FR1	go to MAF.com	-	Wines have price	Pass	-
FR1.5a	FR1	Web services up/down scenario	The web services are running and up	All wines from both suppliers are seen	Pass	-
FR1.5b	FR1	Web services up/down scenario	The first web service is down	Wines only from the second web service are seen	Pass	-
FR1.5c	FR1	Web services up/down scenario	The second web service is down	Wines only from the first web service are seen	Pass	-
FR1.5d	-	Web services up/down scenario	Both services are down	No wines are displayed, message saying wines not found	Pass	not a FR

4.2 FR2

ID	FR	Description	Inputs	Expected outputs	P/F	Comments
FR2.1a	FR2	go to MAF.com	-	Search field is present	Pass	-
FR2.1b	FR2	go to MAF.com	-	Search button is present	Pass	-
FR2.2a	FR2	go to MAF.com	Search for a wine by specific AVIN	Exactly one wine is returned	Pass	result is bound to test data
FR2.3b	FR2	go to MAF.com	Search for a wine by generic description	10 wines are returned	Pass	result is bound to test data
FR2.3c	FR2	go to MAF.com	Search for a wine by black grape	All black grape wines are returned	Pass	result is bound to test data
FR2.3d	FR2	go to MAF.com	Search for a wine by supplier	Wines from one supplier returned only	Pass	result is bound to test data
FR2.3d	FR2	go to MAF.com	Search for a wine by Wales country	All Wales wines returned	Pass	result is bound to test data
FR2.4	-	go to MAF.com	Hit search button with nothing in the search field	All wines are displayed	Pass	-
FR2.5	-	go to MAF.com	Hit search button with random characters	All wines are displayed and message, saying there are no results displayed	Pass	-
FR2.6a	FR2	go to MAF.com	Search for partial AVIN	10 Wines returned with AVIN starting with the query	Pass	result is bound to test data
FR2.6b	FR2	go to MAF.com	Search for partial description	10 Wines returned with description starting with the query	Pass	result is bound to test data
FR2.6c	FR2	go to MAF.com	Search for partial country	All wines are displayed	Pass	result is bound to test data
FR2.6d	FR2	go to MAF.com	Search for partial grape	All wines are displayed	Pass	result is bound to test data

4.3 FR3

ID	FR	Description	Inputs	Expected outputs	P/F	Comments
FR3.1	FR3	go to MAF.com and click on wine's image	-	All wine's detail are displayed	Pass	-
FR3.2a	-	go to MAF.com	look for 'Add to Basket button'	button is present	Pass	not a FR
FR3.2b	-	go to MAF.com	look for quantity field	field is present	Pass	not a FR
FR3.3a	FR3	go to MAF.com and click on wine's image	look for 'Add to Basket button'	button is present	Pass	-
FR3.3b	FR3	go to MAF.com and click on wine's image	look for quantity field	field is present	Pass	-

4.4 FR4 and FR5

ID	FR	Description	Inputs	Expected outputs	P/F	Comments
FR4.1a	-	go to MAF.com and click add to basket on a wine	Basket is empty and invisible	Basket appears with the item in it	Pass	not a FR
FR4.1b	-	go to MAF.com and click add to basket on a wine	Basket not empty and visible	New item appears in the basket	Pass	not a FR
FR4.2a	FR4/5	go to MAF.com, click on wine's image then click add to basket on a wine	Basket is empty and invisible	Basket appears with the item in it	Pass	-
FR4.2b	FR4/5	go to MAF.com, click on wine's image then click add to basket on a wine	Basket not empty and visible	New item appears in the basket	Pass	-

4.5 FR6

ID	FR	Description	Inputs	Expected outputs	P/F	Comments
FR6a.1a	FR6a	Log in, add items to basket, and checkout	Wines from first supplier added only	Order sent only to first supplier	Pass	-
FR6a.1b	FR6a	Log in, add items to basket, and checkout	Wines from second supplier added only	Order sent only to second supplier	Pass	-
FR6b.1	FR6b	Log out if logged, add items to basket, and checkout	-	Redirected to log in page	Pass	-
FR6b.2	FR6b	Log out if logged, add items to basket, and checkout, log in again and confirm that can check out	-	Redirected to log in page, able to log in, and check out then	Pass	-

4.6 FR7

ID	FR	Description	Inputs	Expected outputs	P/F	Comments
FR7.1a	FR7	go to MAF.com	not logged	login and sign up buttons visible	Pass	-
FR7.1b	FR7	go to MAF.com	logged	logout button visible	Pass	-
FR7.1c	FR7	go to MAF.com	logged, then logout	once logged out, login button visible	Pass	-
FR7.2a	FR7	go to MAF.com	not logged, press sign up	redirected to registration page	Pass	-
FR7.2b	FR7	go to MAF.com	not logged, press sign up	registration page asks for name	Pass	-
FR7.2c	FR7	go to MAF.com	not logged, press sign up	registration page asks for email	Pass	-
FR7.2d	FR7	go to MAF.com	not logged, press sign up	registration page asks for address	Pass	-

5 Self-evaluation

Overall I give my self an excellent mark. I have conformed to all Functional Requirements, while providing an OK interface to the MAF user.

What I found the hardest was implementing the MAF application itself. I had no prior knowledge of Rails, so the learning curve was quite steep. I redid every workshop twice or thrice, before I starting working on the MAF project. Ruby was also something to learn, but I did get a grip over it after few days. Rails has way too many things to tinker with, before getting it to do what you actually want. I had to memorize under what folder I can find what. Once I learned Rails, I realized how easy it is to create a simple Web Service that can provide resource as a json data. It took

me several commands to achieve, plus some configuration on the service. I think it was too easy.

Before starting this assignment, I thought I knew what MVC was, and I kinda did. But now reflecting on the knowledge I have acquired, I fully realize what MVC does. In Rails, it is really convenient way of separating the logic of what is seen on the screen, what is in the database, and via which http request a certain action occurs involving the data and the view.

I learned many useful things whilst doing this assignment. The list includes, but is not limited to : AJAX, jQuery, bootstrap, CoffeScript, SCSS(SASS), cross-side scripting attacks and others. What I enjoyed doing the most is adding AJAX based interaction to my website. Not having to redirect the entire page, but only commanding the browser to do something behind the scenes, using JavaScript, was amazing. I found the results quite appealing, both in terms of user experience, and efficiency using the http communication. Bootstrap gem was very good to style my application. jQuerty and CoffeScipt felt so good and easy to use. So much work done, with minimal amount of code.

What	**Comments**	**Deserved mark**
Screencast	The screen cast clearly shows the MAF app running along with two web services.	10/10
Design as reported in the documentation	My design documentation reflects the actual deign behind my MAF implementation fully and correctly. I have added comments to all source files I have worked on (apart from tests). My design conforms to all 7 Functional Requirements.	19/20
Implementation: MAF application	The applications runs. Implementing the MAF was the hardest part. This could easily be turned into Major Project. I believe my MAF is very good. I have several TODO's in my comment indicating things I've though I could further improve given more time.	23/25
Implementation: Supplier web service	It does run. Commented what was appropriate to do so. I believe I am using REST correctly.	12/15
Testing	Great amount of Rails controller/model tests. No cucumber tests though. Didn't have the time for them.	12/15
Evaluation	Evaluation indicated. Said what I found hard/easy.	5/5
Flair	Implemented plenty of actions using AJAX. I use background thread to continually check if wines need refresh. I do refresh only if the ETag of the wines has changed. Great amount of controller tests.	5/10

6 Environment and acknowledgments

- Operating System (OS) - Linux Mint 17.2 (Rafaela)
- Ruby version - `ruby 2.2.3p173 (2015-08-18 revision 51636) [x86_64-linux]`
- Rails version - Rails 4.2.5
- Web browser - Mozilla Firefox 42.0, including SQLiteManager and Firebug plugins
- Apart from the default gems, I have used the following gems:
  - devise
  - will_paginate
  - httparty
  - twitter-bootstrap-rails
  - jquery-ui-rails
- The images used in my application, located in app/assets/images, have their acknowledgments declared in acknowledgment.txt file in the same folder.