BitFX Project Report

NCI Web Technologies Masters 2014

Advanced Rich Internet Applications

Karen Rooney 95004703

Shane Arrowsmith 13109413

Mark Ryder 09107037

Introduction

BitFX was originally conceived as a means to combine an interest shared in the technologies surrounding rich internet applications (RIAs) and the boom currently occurring in digital and crypto currencies, such as Bitcoin and Litecoin. The application is very simple on paper but unique in a number of ways - some of which have made the application very difficult to build and to term either a success or a failure. As background we should also state it was suggested we build this application by a member of the Dublin Bitcoin meet-up group. BitFX will remain an ongoing project and at present represents a very rough prototype of how this application will work and be built.

Project Scope and Background

BitFX was to be an application that would compare trends in traditional currencies and digital currencies. We would supply standard currency conversion charts the type you would see in any bank and then a myriad of different charts based on historical data, that the user would have the option to generate.

Future iterations of this project will see other functionality included such as, a digital wallet to store the currencies and a weekly mailout to customer informing them of the current value of their holdings. Given the scope of the project brief we decided to leave these modules out and concentrate on those parts of the project that focussed on the client side of the application.

This proved to be a difficult build for a number of reasons, some technical and some relating to the infancy of the technologies and knowledge available in the area surrounding digital currencies. We will have the opportunity to discuss the technical issues later but first we will attempt to explain digital and crypto currencies.

Digital and Crypto currencies

Digital and crypto currencies are terms that are essentially interchangeable and have become highly news worthy in recent times and there are many of them. These terms can also include *"a virtual currency, electronic money, or acryptocurrency because cryptography is used to control its creation and transfer."* Originally this was an alternative way of paying for goods on the internet, no different from PayPal. The fact Digital coins didn't need to be backed by "real" money, like in PayPal's case, meant that they have become a commodity and are traded as normal currencies and fluctuate based on market forces. They are thus referred to as decentralised.

Bitcoin is the most recognisable name when it comes to these currencies but others include LiteCoin, Rand, Ripple, PeerCoin and DogeCoin.

'Bitcoins are created by a process called mining, in which users who offer their computing power, verify and record payments into a public ledger in exchange for transaction fees and newly minted bitcoins.Bitcoins can be obtained by mining or in exchange for products, services, or other currencies.'

Crypto-currencies as a form of payment for products and services have seen remarkable growth among internet consumers and sellers. Sellers are generally becoming more inclined to accept digital currencies as they offer lower transaction charges on sales saving them money. World banks have issued numerous warning that digital currencies don't offer the same protections and controls of normal currencies but their growth has continued unabatedly. They can be stolen and don't offer any insurance, worrying especially when considering their popularity among stock and currency exchanges and the price inflation and volatility already experienced around them.

But, with Bitcoin ATM machines starting to appear in cities around the world, the idea of decentralised currencies appears to be more than a brief fad and technologies are starting to grow around them. BitFX aspires to move into this space in the future.

Below we have included a table of the 20 most popular crypto-currencies based on their popularity across the top three global social networks. The numbers relate to their follower numbers across the networks. (*Figures from Crypt.la correct as of March 1st 2014*)

Name	Reddit	Facebook	Twitter
Bitcoin	110186	259598	40400
Dogecoin	68159	58832	80800
Litecoin	19657	4939	13000
ReddCoin	1311	3581	31200
Vertcoin	2720	2074	29800
Quark	1692	13609	1871
Cryptogenic Bullion	132	16202	735
Pandacoin	43	9449	766
Potcoin	1331	6706	1847
MaxCoin	515	84	8709
Nxt	309	7346	1033
Catcoin	422	6842	1119
Fastcoin	36	448	7828
Megacoin	436	7037	834
Ripple	757	888	5302
Feathercoin	524	1512	4546

Name	Reddit	Facebook	Twitter
FedoraCoin	1068	3993	379
Peercoin	3045	939	1355
EarthCoin	241	4437	630
Primecoin	1554	876	1731

Area of contribution

As outlined previously the major aim of this iteration of the project was to become comfortable with some but not all areas of the proposed application. If you explore the myriad of websites and applications dedicated to currencies you will quickly become bored as they are quite dense and off little to the casual user. "Mining" and trading digital coins has become a hobby among many of the world's tech savvy young professionals. Thus we aspired to create an application that is simpler and a more visual than the standard currency website. To this end the idea of creating a RIA was very appealing.

As a RIA project it was prudent to concentrate on the presentation layer and explore some of the technologies associated with JavaScript frameworks, CSS3 and responsive design. With this in mind the project became concerned with the following areas and the following requirements were decided upon.

1- Currency information:

   We wished to display the current exchange rates of currencies which would update each time the page was called and would give the data some context by also pulling in yesterday's exchange rate average.

   We started to research APIs and found it quite difficult to find any affordable APIs with all of the digital currencies we wished to use listed. As mentioned earlier, for this development iteration we were more interested in the process so we found an API at openexchangerates.com which was quite reasonable to developers and decided to use that. It had 3 of our digital currencies listed along with 180 other world currencies.

   To achieve this we would need to make 3 API calls per page and then present the data. We decided that we would make use of a JavaScript framework to organise our API calls.

2- We wished to provide users with a way of interacting not just wish today's data but with data historically too. Users would be able to see a visual representation of the performance of a currency over a week, a month or a quarter.

We would use a chart plug-in for this. During early tests for this functionality it became apparent that we would need to provide a way to make our data persist. The API we used was quite flexible when it came to historical data but it would take almost 100 calls to populate a table, which was impossible for us to accept. Thus we also decided to build our RIA upon a Ruby on Rails backend to avail of a database and create our own API from data we had collected.

The decision to use Rails, and specifically its conventions, would ultimately lead to many of the failings of the project. We will discuss this in greater detail as we move through the project report, at each point we have tried to be both fair and critical of how our approaches led to both the successes and failures of the final app.

State of the Art Review

To reiterate what was alluded to earlier, BitFX would be using a number of technologies to try and achieve our goals. This would include the use of Ruby on Rails for the application framework, AngularJS to manage the calls from and responses to the API. The calls would be returned as raw JSON objects so would need to be transformed into datasets, Angular would then aid with the presentation of data. Twitter Bootstrap would be used to create the responsive interface through a mix of SaSS and BootstrapJS. RSpec and its descendent Jasmine would be our testing frameworks. Where possible we would also try to use modern CSS3 transitions to remove some of the scripting burden, along with HTML5 elements on the page.

AngularJS

AngularJS is an open-source JavaScript library managed by Google. To explain it in their words;

'AngularJS is a structural framework for dynamic web apps…. let's you use HTML as your template language and…. extend its syntax….. Angular's data binding and dependency injection eliminate much of the code you currently have to write. And it all happens within the browser, making it an ideal partner with any server technology.'

To elaborate, Angular allows us to give an MVC -or as it refers to itself, Model-View-Whatever - architecture to DOM manipulation. With the shift in computing power towards the client Angular gives us more control, structure and modularity when approaching many scripting problems. As developers, we are usually familiar with these patterns of coding when it comes to server side frameworks, but using these concepts on the client side proved a little challenging. It is easy to see the power of Angular even if BitFX merely scratches the surface of its potential, as we become more comfortable with the framework we will include more features.

For now we were satisfied to explore some of more obvious benefits of Angular such as;

"Minimising the impedance mismatch between document centric HTML and what an application needs by creating new HTML constructs. Angular teaches the browser new syntax through a construct we call directives. Examples include:

- *Data binding, as in {{}}.*
- *DOM control structures for repeating/hiding DOM fragments.*
- *Support for forms and form validation.*
- *Attaching code-behind to DOM elements.*
- *Grouping of HTML into reusable components."*

As Orbitone point out, Angular also makes the code-base smaller and shorter when compared to other Frameworks which improves code stability and improves maintainability. This means the code will be less susceptible to bugs and is a lot easier to test. Code modularity promotes reuse, as it is in other frameworks. For example, API calls all follow the same structure and are easily portable to other projects.

```
var app = angular.module('myApp', [ ]);

app.controller('currencyList', function ($scope, $http){
  $scope.currency=[];

  $scope.getCurrencies = function(){
    $http({method : 'GET',
                url :
      "https://openexchangerates.org/api/currencies.json?app_id=6334fc4be1884035840275a
      de9f84c9a"})
      .success(function(data, status){
        $scope.currency = data;
      })
      .error(function(data, status){
        alert("error");
      });
  }
});
```

In the above example we have a basic Angular API call taken from the BitFX source code. In BitFX there are three API calls made to populate the tables presenting the current exchange rates to the user, all three APIs follow a similar structure.

First we see the Angular application being declared as a module, with the App name and an empty array being passed to it to represent the dependencies that will be passed to the application. The empty array will hold the responses from the three functions making and interpreting the API calls. The reason we pass these to an array is because we have multiple controllers but can only declare an Angular module once on the page. The array allows to pass multiple dependencies to the application, in more complex applications these can include routes and factories also.

Next we declare the API call as a controller. In this case the API was returning a list of currencies. We pass to variable to the controller, $scope and $http which have very specific meanings in terms of Angular. In Angular they are actually referred to as services rather than variables as they are built in and reference powerful functions in the Angular framework.
Here scope represents our data model, and is essentially holding the API return data until the view is ready to present the data:

"Scope is the glue between application controller and the view. During the template linking phase the directives set up $watch expressions on the scope. The $watch allows the directives to be notified of property changes, which allows the directive to render the updated value to the DOM."

The $http service is a core Angular service that facilitates communication with the remote HTTP servers via the browser's XMLHttpRequest object or via JSONP. We then create another function with $http called "getCurrencies", that calls the API data, returned here as a JSON object and pushes it to an array to make it easier to access in the views.

As for presenting the data, Angular make this a very straight forward task.

```
<div ng-app="myApp">
<div class="divLeft" ng-controller="currencyList" ng-init="getCurrencies()">
 <table class="table table-striped table-bordered table-condensed">
  <th>Currency</th>
  <tr ng-repeat="(key, data) in currency">
  <td>{{data}}</td>
  </tr>
 </table>
 </div>
```

First we declare the module we wish to access via the ng-app call. Then we call the controller we wish to access and call the getCurrencies() function. Here we have decided to present the data in a table. On the table row we ask Angular to repeat the data for the length of the array we push the API data to. On the table data, we tell angular we only wish to display the data in the key-data pairs from the array with the {{data}} tag. These {{}} tags will be familiar to those who understand PhP as they represent data binding, they perform a similar task.

As you can see Angular is a very powerful framework with enormous potential to improve the client side of an application. BitFX hasn't used this to the complete extent of its potential and we will discuss this later during the critical analysis of our approach. We also had a number of difficulties with using Angular within a Rails application, some of our own making.

Twitter Bootstrap and Rails Bootstrap SASS gem

Twitter Bootstrap is a free library of tools used to make static and responsive UI design easier within web applications. It contains a number of HTML and CSS-based design

templates for all the standard elements that go into creating web interfaces, as well as JavaScript extensions that make responsiveness much easier to incorporate into an application. As of March 2014 it is the most popular project on Github and is quickly becoming the industry standard in terms of UI design and development.

Originally Bootstrap was built as an internal tool for Twitter employees to speed development work across their departments. In 2011 it was released as an open source project and on its third version is now compatible with all modern web browsers. Bootstrap is completely modular and built upon a series of LESS stylesheets. It has become popular among developers as we can select those components we wish to use.

In their article “11 reasons to use Bootstrap”, site-point explain the major advantages to using Bootstrap including, time saving, customisation, consistency of design, future compatibility and the fantastic quality of the documentation available to developers. But rightly they point out that the major advantages are in the realm of design, layouts and responsiveness.

“The Bootstrap libraries offer readymade pieces of code that can pump life into a website.”

```
<div class="navbar navbar-fixed-top">
 <div class="navbar-inner">
  <div class="container">
   <a class="btn btn-navbar" data-toggle="collapse" data-target=".nav-collapse">
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
    <span class="icon-bar"></span>
   </a>
   <a class="brand" href="#">BitFx</a>
    <div class="nav-collapse">
     <ul class="nav">
      <li><%= link_to "Home", root_path %></li>
      <li><%= link_to "Current Rates", current_rates_path %></li>
      <li><%= link_to "About", about_path %></li>


     <% if signed_in? %>

      <li><%= link_to "Users", users_path %></li>
      <li id="fat-menu" class = "dropdown">
       <a href="#" class="dropdown-toggle" data-toggle="dropdown">
       Account<b class="caret"></b>
       </a>
       <ul class="dropdown-menu">
        <li><%= link_to "Profile", current_user %></li>
        <li><%= link_to "Settings", edit_user_path(current_user) %></li>
        <li class='divider'></li>
        <li>
         <%= link_to "Sign out", signout_path, method: "delete" %>
        </li>
```

```
        </ul>
       </li>

      <% else %>

       <li><%= link_to 'Sign in', signin_path %></li>

      <% end %>

      </ul>
     </div>
    </div>
   </div>
  </div>
```

Above we see the code for the navigation bar common across all BitFx pages. If we ignore the embedded Ruby code and look as the Bootstrap features it is easy to highlight the advantages available to developers. By adding a selection of standard of Bootstrap class names to elements we have created a responsive navigation for the site.

According to the Bootstrap documentation, "navbars are responsive meta components that serve as navigation headers for your application or site. They begin collapsed (and are toggleable) in mobile views and become horizontal as the available viewport width increases." In other words the navigation bar resizes or collapses itself depending on the device it is viewed on.

Firstly we declare the type of navbar we want. In this case we have gone for a "navbar-fixed- top" which mean the navbar will be fixed to the top of the browser window and always be visible when scrolling down the page. This navbar will also overlay other content on the page so padding must be added to the body container in CSS. Bootstrap then make it simple for us to add extra features to the navigation. The anchor tag containing the "btn btn-navbar" class refers to the button that will hide and unhide the navigation menu when viewing the page on smaller screens such as mobile devices.

Another advantage of Bootstrap is the fluid grid system available when laying out content. The grid system is responsive and scales up to 12 columns depending the device size. This system make laying out pages simple without having to worry overly about responsiveness when viewing on other devices. Within BitFX there are examples of this method of laying out content among other Bootstrap standards. For another example, below there is a way of creating beautiful responsive tables, used to display the data returned from our API calls. This is all made possible by adding the 4 class names below. Another advantage of Bootstrap is that you can chain commands and classes together and use them in tandem with each other.

<table class="table table-striped table-bordered table-condensed">

As outlined above, Bootstrap uses LESS stylesheets to achieve many of its templates. But as it is open-source its functionality has been extended to meet many ends. BitFX is built upon a Ruby on Rails backend, it thus seemed prudent to try to persevere with many of the

conventions of the framework. Rails allow you to use any stylesheet and JavaScript language or library you wish. The preferred stylesheet language of the Ruby On Rails is SASS.

SASS stands for Syntactically Awesome Style Sheets and is a scripting language that is fully compatible with CSS, and interpreted into CSS by the browser. SASS essentially extends the functionality of CSS and provides us with a number of features more commonly associated with Object Orientated Languages, such as variables, Booleans, nesting, modularity and inheritance.

Below is an example of how we can assign a hexadecimal colour reference to a literal colour name via the $ variable declaration. This means we only write the hex value once and then can refer to it repeatedly throughout the styelsheet by any name we have assigned.

```
$grayMediumLight: #eaeaea;
```

Below is a uniquely SASS concept. That of a mixin. "A mixin lets you make groups of CSS declarations that you want to reuse throughout your site. You can even pass in values to make your mixin more flexible. A good use of a mixin is for vendor prefixes. " This saves us rewriting the code for different browsers when it comes to browser specific commands such as border-radius or box sizing.

```
@mixin box-sizing {
        -moz-box-sizing: border-box;
        -webkit-box-sizing: border-box;
        box-sizing: border-box;
}
```

Like Less, SASS also allows us the luxury of nested classes and ids. For example in the BitFX footer we are able to declare all of the styles and positioning related to the footer element. Within the same SASS block we can also declare the styling for all of the child elements contained in the footer by merely nesting the elements in relation to their position in the DOM hierarchy.

```
footer {
        margin-top: 45px;
        padding-top: 5px;
        border-top: 1px solid $grayMediumLight;
        color: $grayLight;
        a {
                color: $gray;
                &:hover {
                        color: $grayDarker;
                }
        }

        small {
                float: left;
```

```
        }

        ul {
                float: right;
                list-style-type: none;

                li {
                  float: left;
            margin-left: 10px;
                }
        }
}
```

Also available in SASS but not explored in BitFX is the potential to create CSS partials for repeating code that can be modularised and called when needed by using the @import command. These can be called anywhere within the structure of the SASS script. We also use @import to call the Bootstrap dependencies into the project. It is important to point out that if you wish to override any of Bootstrap's default styles it needs to be done before the @import. The nature of SASS is to take anything before the import as default and unchangeable so if a developer wished to override Bootstrap this is how it would be done.

Within BitFX it was decided to install the SASS version of Bootstrap and to use the conventions of the RoR framework. Using this version of Bootstrap required the installation of a Ruby gem. Gems are package managers within Ruby for extending functionality and distributing Ruby libraries, programs and extensions. Once installed the Bootstrap-Sass gem provides us with the source files we need to start making SASS changes and overrides to the bootstrap layouts.

Ruby On Rails

The decision was made early in the project to use a web application framework with the project. As we were all comfortable and experienced within the Ruby On Rails framework it seemed the obvious choice. However ultimately it proved to be the incorrect choice. In retrospect it would have made more sense to have used a smaller project structure as many of Rails major features were irrelevant to a project that aspires to be a RIA. During the critical evaluation of the project we will discuss this in more depth but it should become apparent when describing Ruby On Rails why we made such a poor initial decision.

Ruby on Rails is a web application framework, meaning that it offers the ability to interact with a web server, query a database and render html templates straight out of the box. While these are features and characteristics you would not necessarily expect to find in the average RIA application they represent key features and advantages when developing in the agile methodology, as Rails gives you something to work with faster, a skeleton of an application to add to and enhance.

Rails is a development framework designed to best emphasize the patterns of Convention over Configuration and Don't Repeat Yourself. Outside of the MVC architecture paradigm,

they are the two defining concepts in the nature of Rails. To be critical of our approach to RIA development, it is not impossible to build RIAs on Rails but you need to not forget the principle of convention over configuration - we spent far too much time configuring on BitFX and should have changed direction at vital times rather than trying to change Rails to meet our needs.

Convention over Configuration is a design principle seeking to reduce the number of decisions you make during development which enhances the simplicity of your application without losing the ability to change things. Convention means less time is wasted writing Configuration files as Rails presumes certain behaviour patterns and implements it efficiently and pre-tested. Essentially COC prescribes to the notion that there are repeating problems that crop up in development and that there are tried and tested ways to fix those problems so why look for another solution, just use the one you know that works. This is key to the way Rails works as a framework.

Some of the more disappointing aspects in the finished version of BitFX are not adhering to that rule. To highlight this if we compare the previously mentioned installation of AngularJS and Bootstrap in the project.

Rails has a 2 key features both of which were used to install, use and deploy Bootstrap and Angular. One is known as the asset pipeline, the other is Ruby Gems.

“The asset pipeline provides a framework to concatenate and minify or compress JavaScript and CSS assets. It also adds the ability to write these assets in other languages and pre-processors such as CoffeeScript, Sass and ERB.”

Rails also allows for the use of external file libraries and plugins known as gems. Gems can be used to provide functionality we may desire in our applications such as logins, permissions, payment gateways, forums etc. and can also be used to solve smaller problems or to extend existing functionality through libraries and frameworks such as Bootstrap and Angular.

In the case of both Angular and Bootstrap we used Ruby Gems to make them accessible to the project. In the case of Bootstrap we installed the Bootstrap-Sass gem and wrote all of our styling and overriding of the library in native SASS adhering to the conventions of Rails. In the end the usage of Bootstrap while not perfect was far more successful and useful than Angular.

In the case of Angular we used the AngularJS-rails gem. This gem coupled with the usage of the asset pipeline provided a large challenge throughout development. The asset pipeline works perfectly in development but as soon as the environment is changed to production Rails wants to minify all stylesheets and JavaScript files. With Angular this meant that the AngularJs-rails gem came with a list of dependencies such as the ngmin-rails gem which is needed to handle the minification of Angular code in production. We made a number of

errors trying to get these dependencies right adding extras and rewriting configuration files and ultimately coded ourselves into a corner. Rails is environment, version and dependency specific, which is fine when you work with Rails, we started to fight with Rails conventions and if there is one major failing in the project that is it. We were advised at several junctures to consider migrating the project to a different environment, neglecting to do so was an error.

BitFX represented a huge learning curve in terms of both designing and architecting RIAs but also gave an education to us on the limitations, conventions and advantages of the Rails framework when trying to architect a solution. If we had used Angular with Coffeescript as Rails would prefer, or downloaded the Angular source files and worked with Angular that way we may have experienced a better outcome and spent the considerable time we put into the project in a better way.

High-Charts

We went through a number of data visualisation and presentation tools to decide on which to employ in this project. Many of them such as Raphael.js, Chartjs and Google Charts are really beautiful and free to use for projects such as ours. There were a number of reasons for choosing Highcharts:

- its documentation made it simpler to apply for intermediate users

- it has a great reputation in the developer community, both from talking to colleagues and from feedback on stack overflow

- it is commonly and seamlessly used in Rails (there is a Highcharts ruby gem available)

- it looks well alongside the default Bootstrap styles, and it has cool features such as huge range of graph styles, its responsive, customisable and displays low-level detail in clickable tooltips, animation, zooming.

Our plan had been to employ Highcharts once the angular API calls were working, however given the complexity and difficulties we began to encounter here, and the fact that a great deal of historical data (ergo individual API calls!) would be needed, we took a different approach.

We built an internal API using the Rails framework, with a rates model and controller calling the Open Exchange API, these would store as many Rates records as we wanted to - currently the system is set to call back as far as January 1st 2014, with a method that updates the Rates model if it has not been updated for more than one day.
We included the HTTParty Gem, a wrapper library for working with HTTP API's in Rails. It is loaded in the Gemfile & bundled, then inside our controller we could use it as follows:

```
class RatesController < ApplicationController
 include HTTParty
{...}
 def get_more_data(latest_date)
  (latest_date..Time.now.to_date).to_a.each do |this_date|
   their_response =
HTTParty.get("https://openexchangerates.org/api/historical/#{this_date.strftime('%Y-%m-
%d')}.json?app_id=6334fc4be1884035840275ade9f84c9a")
    if their_response['rates']
     their_response['rates'].each do |ccy_name, the_rate|
      FxRates.create(
       value_date: this_date,
       rate:          the_rate,
       currency:   ccy_name
      )
     end
    end
  end
 end
```

We then created an FX API Controller with an Index and Show method would render the json into the front end. This way we had all the data we needed locally, without thousands of inefficient requests being sent, and it also allowed us to run the charts offline. Highchart is a JavaScript charting library which has an excellent free tier for non-commercial use and that requires JQuery, Mootools or Prototype. We used JQuery, inserting the jQuery min file, along with Rails js and Highcharts js into the vendor assets file and calling them in the application Header.

From here, we set up an additional html(erb) page and used Bootstrap to format the dropdown list and the span of each Chart, with JavaScript calls for the Currency list which appears in the dropdown, and thereafter calls to repopulate three Charts, depending on the currency selected from that dropdown.

The JavaScript chart calls are made directly to the internal API service in the app, setting a ‘make a graph’ function with variables for the div, graph type, number of days and the

graph type, we've used two basic Line Charts and a Bar Chart to represent the currency fluctuations over a week, a month and a 90 day period.

User Interface Design

The interface design within the application can at present only be regarded as incomplete and unsuccessful. To fix these UI issues it was a matter of time getting away from us on other development problems as outlined earlier. The code for all the UI features are presently in the project for review but need a lot of fine tuning before they could be deployed for real world users.

At the outset we wanted to give the application a minimal, modern and information driven design that would offer the user the information they required at a glance and minimal fuss. Due to the changing nature of how users browse the internet on a myriad of browsers and devices we wanted to put an emphasis on responsive design and sleek transitions.

As a result we tried to adhere to the conventions of Bootstrap and not overly override its features or default styling. We decided on a limited colour palette of blues, greys, blacks and oranges as they all work well together. As designers we are all wary of images for images sake so we decided to keep them to a minimum and let the charts and tables take all of the attention from the user.

There are a number of other elements and experiments throughout the site that aspired to enhance the user's experience of the site and give a general air of professional design. Also this project represented a chance to put into practice many of the new technologies we have been exposed to over the course. With that in mind we made attempts to explore some CSS3 features and JQuery UI in some of the lesser pages where the technologies would not interfere with the larger functionality of the application.

On the about page there is an example of CSS3 transitions. CSS3 has all the presentation abilities of standard CSS but has a number of new elements all of which make the development of effective UIs easier and involve a lot less scripting. The major improvements have been influenced by a lot of the CSS extenders such LESS and SASS. With very little CSS we can now animate, replace content, transform elements in 2D and 3D, add text effects, traverse the DOM easier by way of more accurate selectors and handle the whole area of transitions more efficiently. Transitions are an area we were interested in designing the UI for BitFX.

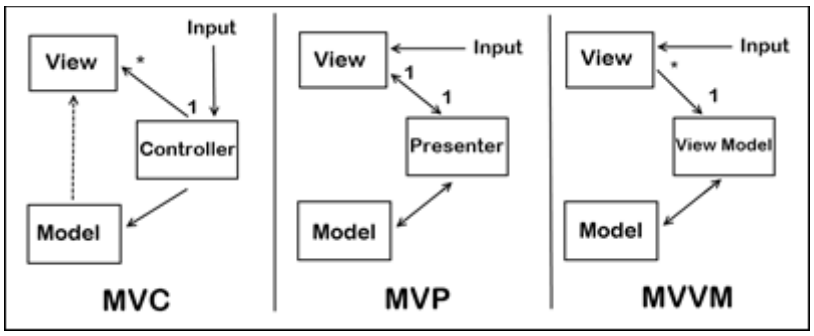
Transitions allow us to add effects and animations when moving from one style to another without the need to use the traditional handlers of these features such as Flash or JavaScript. By using these transitions, along with Twitter's excellent Font Awesome icon library, we were able to create a responsive, interesting and useful about page that at first glance looks like three images. Rolling over the images a nice transition occurs allowing the user to social connect with the designers of BitFX.

The home page features a similar if slightly more complicated transition aided by JQuery UI were clicking the central image sees the image transition into a useful navigation for the user. These are all professional looking elements built to enhance the user's experience of the application.

We have previously discussed how Bootstrap and High Charts were used to enhance and interpret the Data retrieved from the API and presented to the user. Much of the success of the UI is down to following the standards and conventions of Bootstrap aided by brilliant templates and documentation available on the Bootstrap Github page.

Application Architecture

The architecture of BitFX was dictated by the tools chosen to create the project. Within the project there are two versions of MVC in use, one pertaining to AngularJS and the other too Ruby On Rails. Both implementations are slightly different but follow the same basic principles.



Both Rails and AngularJS stick to the MVC architecture pattern. It is unadvisable to go against the conventions of MVC when developing in either. This pattern divides an application into three constituent parts and separates the representation of information and logic from how it is interacted with by users:

1. Model – This contains the business logic and application database of our system.

   Within BitFX we did not wish to make hundreds of API calls each time a user wanted to interact with the charts service. It was obvious that in order for data to persist we would need a database of some sort, so we availed of Rails MVC. From the historical data available from our API we were able to store data locally and then create our own API to populate and maintain our chart functionality.

   Within Angular the model is more concerned with, and defined by, the data retrieved from the API calls. If, as in our case, the API is vending JSON, then these objects will already be 1st class JavaScript objects. As previously mentioned the scope is used for carrying the data between all pieces of the system.

2. Controller – mediates on input and output, generating commands for the Model or the View.

Within BitFX there are examples of how both Angular and Rails treat controllers. As the application is built with the Rails Framework every page that is served needs some kind of controller. Rails comes with a built in routing system which links any user requests to code via a controller. So even though the majority of the code and logic being presented in the application was client-side, within Rails for the page to be served there still need to be at least a reference to it in the controllers. There is also some interesting controller logic when where we created the API to populate the charts.

Within Angular, controllers are much more powerful than within Rails. In Rails we are always advised towards "Thin Controllers, Fat Models" meaning that you should try keep as much of your applications logic as possible abstracted in models and helper functions and merely call them into the controllers as needed. It is part of the DRY (Don't repeat yourself) principle used in RAILS.

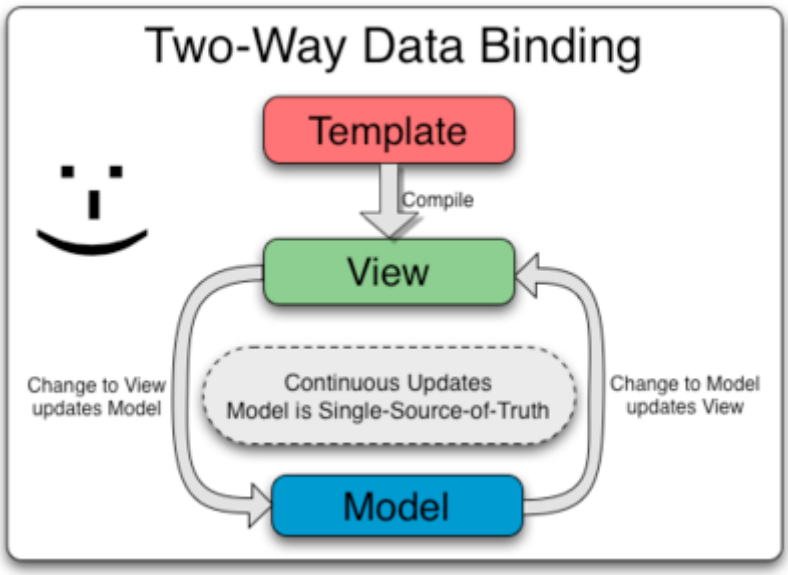
While Angular refers directly to controllers, when discussing angular controllers you are also referring to the range of routes, services and factories available when modularising your code. Thus controllers are very powerful. Controllers can query the REST APIs and makes available what is necessary for the View on the $scope. Provide callbacks for directives to respond to events that might then require calls back to the server. Angular controllers can also handle validation - usually via a callback to a directive. You don't want your user to wait for the server to validate everything - the client should know *something* about the validation to give the user immediate feedback.

To emphasise the difference between Rails and Angular controllers, Angular would expect the business logic to reside in the controllers. Rails would expect it to reside in the model. As such Angular's architecture while starting from an MVC position is often termed as MVVM – Model-View-View-Model - or MVW – Model View Whatever.

Due to BitFX being our first attempt at using Angular in an application this was the concept that we struggled with most. As a result we purposely left the Angular code we used very basic. Currently we have 3 API calls which each pass the data retrieved to the scope for presentation in the view. In respect to future projects, this now seems like an inefficient method of making API calls and we should have made the code more modular. We could have created either a service or a factory to handle the calls and used our controller solely for the manipulation of data which in turn would have made our presentation a lot smoother. As it stands the fact we have three controllers, one for each API call made the presentation of data nearly impossible and gave us a disappointing result in terms of our UI.

3. View – the output representation of data.

Rails and Angular views are very similar and interesting as they also offer embedded code and data binding, with which we can perform simple logical operations on the page with data received from the controller in either framework.

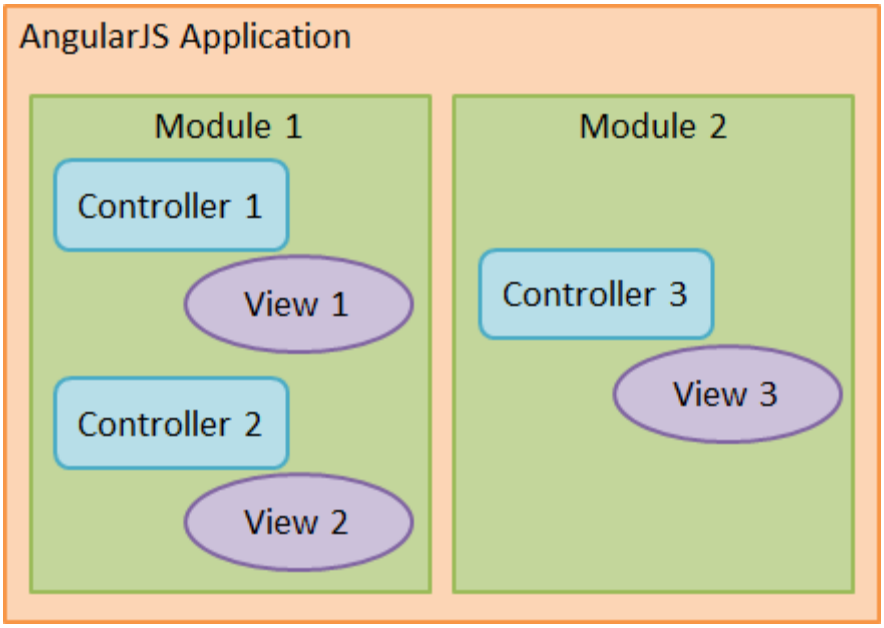


While Angular and Rails maybe differ in the area of controllers they are similar in that the MVC pattern defines the interaction between each constituent part of the frameworks.

In Rails the controller responds to external request from the web server to the application or model, and then also decides which view file to render. The controller may also directly query the model and pass this back to the view. A controller will typically provide one or more actions or methods. With Rails this gives rules as to how to respond to specific requests. The controller is aided through all of this by the Rails routing system. If the route is not mapped the controller will not be accessible to web requests. The controller actions generally follow RESTful routes; create, new, edit, update, destroy, show and index. REST works on the idea that instead of relying exclusively on the URL to determine what page you want to go to, a mixture of action and URL can get you to several different pages related to the URL depending on the action appended to the URL. Thus a URL, when used with a RESTful action (get, post etc..) will navigate to a different page.

Within Angular controllers are more concerned with the manipulation of received data or DOM events. It is still possible to make Angular controllers as powerful as Rails controllers by creating cooperating services, factories and routes. But while Angular controllers have the power to create and manipulate DOM elements, they don't yet have the power to serve pages – that is not what it is designed for anyway.

As previously stated MVC is the preferred architecture paradigm within Rails. Thus Rails has many useful inbuilt features to create the basic architecture and ensure we stick to it. For example when you start a new application with the "rails new project" command, Rails automatically sets out a file system and hierarchy best suited to follow the MVC way. With Angular we didn't have this luxury and a certain amount of inexperience with the Framework means we never really got to grips with best practice within its architecture.



So if we are to be critical of ourselves in terms of our architecture it is in not fully understanding the way the Framework wants you to develop. That was a key error and led to a lot of the mistakes made during development and fed into the apparent simplicity of the application. We are all experienced Rails developers so by default we understand MVC architecture and DRY coding principles. We were constantly on the right track in this development cycle but never quite got to grips with Angular. Time constraints meant we were constantly firefighting and hacking together elements we thought would work. The reality is, if we had stepped back and noticed the similarities between the two frameworks earlier we would have had a much more successful app by mapping the knowledge of architecture patterns and frameworks we already had onto Angular.

Data Transfer Strategies

As outlined on page five through to seven, Angular JS is used for the data transfer of the currency rates. The currency rates are located on a JSON API on openexchangerates.org. The Angular JS script, called marketList.js, creates a controller which uses the HTTP method GET to pull the JSON API into the BitFx application and renders the data on to the page view, current_rates.html.erb. Unfortunately all the data needed to complete the page view were not included in the one API. This means multiple API calls were required from the marketList.js which involves creating a separate controller for each API call. To call an Angular controller in a html page you would normally use

<div ng-app ng-controller = “nameOfController()” >

However you can only use ng-app once on a html page. To overcome this problem, all the controllers required for the html page are placed inside an Angular module in marketList.js. To call the controllers the code now looks like

Now all the data needed for the current rates page can be rendered.
We could not find any API that contained data for the country flags. Instead a gem was used that contained all the international flags of the world. The problem with this gem was that it contained flags we didn’t need so we couldn’t loop through the gem to pull out all the flags. We had to go through the gem find the code for the flags we needed and hard code them into the page next to the currency table ourselves.

Evaluation and Testing

Ideally we would have liked to employ a number of usability tests on to the application. Usability and good UI design are some of the defining characteristics relating to Rich Internet Applications. Early on we did design wireframes (included in the application file base) and had a primary persona in mind for the application.

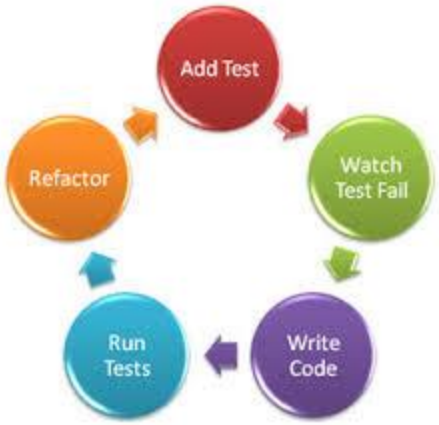
The primary persona was originally to be a young technically savvy professional with an interest in digital currencies. BitFX will be rebuilt in line with the original requirements and with this persona in mind over the coming months.

Unfortunately, BitFX never got close enough to completion over this development cycle for proper prototypal usability testing and as such it was impossible to evaluate in this way. Had we gotten to that stage it would have been prudent to test the application for the four major usability principles.

- Perceiveability
- Operability
- Understandability
- Robustness

At present BitFX may pass (!) in terms of perceiveability and understandability but would definitely fall down on the other two. With such obvious flaws in the architecture and completion of the project there was little need to evaluate the app under these terms. In future, with an improved project this type of testing will be essential as RIAs aspire to user experience and as such it is as important as testing the code base.

Jasmine, Rspec, TDD and BDD



During the development cycle of BitFX there was an attempt made to use test and behaviour driven development patterns within the application.

Test-driven development is a software development process that relies on the repetition of a very short development cycle, easily memorable by the phrase red-green-refactor. First a developer builds a test case that defines the objective of the code or function, this should initially be a failing test (RED). Rudimentary code is then written with the sole objective to pass that test (GREEN). Once we have a passing test the code is refactored to the standard desired by the developer safe in the knowledge it works.

Behaviour-driven development is based on test-driven development (TDD). It *"combines the general techniques and principles of TDD with ideas from domain-driven design and object-oriented analysis and design to provide software developers with shared tools and a shared process to collaborate on software development, with the aim of delivering "software that matters"……the practice of BDD does assume the use of specialized software tools to support the development process….. The tools serve to add automation to the ubiquitous language that is a central theme of BDD."*

With this in mind there are a number of interesting tools used in BitFX towards making the testing process easier. We will try to describe some of them now.

RSpec and Jasmine

Rspec and Jasmine is sister technologies providing a behaviour-driven development (BDD) framework for both the Ruby and JavaScript programming languages respectively. They contain their own mocking framework that is fully integrated into the Rails framework. As such as testing frameworks they can be *considered "domain-specific language (DSL) and resembles a natural language specification."* Explained simply, this implies that Rspec adheres to the coding conventions of Ruby and Jasmine adheres to the conventions of JavaScript. Jasmine is actually based on Rspec which makes learning and understanding one of them as simple as understanding the other.

```
describe('Foo', function() {
 describe('blah', function() {
```

```
  it("does something", function() {
    expect(1 + 1).toBe(2);
            });
  });
});
```

Above is a sample jasmine test, it is immediately apparent how it resembles native JavaScript. It merely describes the function it wishes to test and the expected outcome of the function. Jasmine gets more complicated with the matchers it uses in more complex functions but it is a great way to simplify your codebase and be more focused when writing you code.

```
describe User do

            before do
              @user = User.new(name: "Example User", email: "user@example.com", password:
"foobar", password_confirmation: "foobar")
            end

            subject { @user }

            it { should respond_to(:name) }
            it { should respond_to(:email) }
            it { should respond_to(:password_digest) }
            it { should respond_to(:password) }
            it { should respond_to(:password_confirmation) }
            it { should respond_to(:remember_token) }
            it { should respond_to(:admin) }
            it { should respond_to(:authenticate) }

            describe "remember token" do
                        before { @user.save }
                        its(:remember_token) { should_not be_blank }
            end

            it { should be_valid }
            it { should_not be_admin }

            describe 'with admin attribute set to "true"' do
                        before { @user.toggle!(:admin) }

                        it { should be_admin }
            end

            describe "when name is not present" do
                        before { @user.name = " " }
                        it { should_not be_valid }
            end
```

```
        describe "when email is not present" do
                before { @user.email = " " }
                it { should_not be_valid }
        end

        describe "when name is too long" do
                before { @user.name = "a" * 51 }
                it { should_not be_valid }
        end

        describe "when email format is invalid" do
                it "should be invalid" do
                  addresses = %w[user@foo,com user_at_foo.org example.user@foo.
foo@bar_baz.com foo@bar+baz.com]

                  addresses.each do |invalid_address|
                    @user.email = invalid_address
                    @user.should_not be_valid
                   end
                end
        end

        describe "when email format is valid" do
          it "should be valid" do
            addresses = %w[user@foo.COM A_US_eu@f.b.org frst.lst@foo.jp a+b@baz.cn]
            addresses.each do |valid_address|
              @user.email = valid_address
              @user.should be_valid
            end
          end
        end

        describe "when email address is already taken" do
                before do
                        user_with_same_email = @user.dup
                        user_with_same_email.email = @user.email.upcase
                        user_with_same_email.save
                end

                it { should_not be_valid }
        end

        describe "email address with mixed case" do
                let(:mixed_case_email) { "FoO@ExAmple.CoM" }

                it "should be saved as all lower-case" do
                        @user.email = mixed_case_email
                        @user.save
                        @user.reload.email.should == mixed_case_email.downcase
                end
        end
```

```
        describe "when password is not present" do
                before { @user.password = @user.password_confirmation = " " }
                it { should_not be_valid }
        end

        describe "when password_confirmation is nil" do
                before { @user.password_confirmation = nil }
                it { should_not be_valid }
        end

        describe "With a password that's too short" do
                before { @user.password = @user.password_confirmation = "a" * 5 }
                it { should be_invalid }
        end

        describe "return value of authenticate method" do
                before { @user.save }
                let(:found_user) { User.find_by_email(@user.email) }

                describe "with valid password" do
                 it { should == found_user.authenticate(@user.password) }
                end

                describe "with invalid password" do
                 let(:user_for_invalid_password) { found_user.authenticate("invalid") }

                 it { should_not == user_for_invalid_password }
                 specify { user_for_invalid_password.should be_false }
                end
        end
end
```

Above we can see the similarities between Rspec and Jasmine. Rspec is written to resemble Ruby whereas Jasmine is written to resemble JavaScript, but the basic principles remain. The above script is a series of tests for the log in and registration system within BitFX. It is always prudent when dealing with forms and registration data to have validation on both the server and client side for security reasons. The above script tests the possible series of events we expect to happen when a user goes through this routine. We want to ensure that the user model is validating the data correctly.

Similar to the jasmine functions we see a lot of the same language used - 'describe', 'expect', 'before', "toBe". It is so easy and useful to use these frameworks.

There are also a number of other dependencies to point out within the project relating to how we set up out testing framework:

- Factory_Girl: This is a helper gem used to provide a framework and DSL for defining and using factories - less error-prone, more explicit, and all-around easier to work with than fixtures. Within the application we use this gem to create a fake user

when running our tests, we can have a reference to a user, a password and any other details we wish depending on what we need to test.

- Faker : Is used to populate a database with fake users should you need to perform scalability and stress testing.
- Capybara: Capybara is an integration testing tool for rack based web applications. It simulates how a user would interact with a website.
- Spork – Spork allows us to have a test DRB server running at all times. These types of servers are faster when running tests as there is now need for TCP sockets.

Admittedly there is a bit more in the way of Rspec testing in BitFX, than Jasmine. That is not to say we can't test what is happening in a RIA application. One of the major advantages of RSpec is that you can implement an MVC pattern to the tests also. Thus given a series of events you can expect certain elements or events to happen on a view at certain times. It maybe isn't as clean, efficient and accurate a way of running TDD with JavaScript but it does provide us with some insight. In future iterations we will need to stick more rigidly to Jasmine for JavaScript or Teaspoon as we expect to use Angular with Coffeescript over the next iteration.

Summary

On submission of this project, we have not managed to achieve the goals we had set out to. Although we have put some good functionality together and written code and tests used some really cool frameworks and libraries and both built and used an API - we encountered too many compatibility problems which we were unable to resolve. The current rates page, using Angular JS, works on one laptop but not on the rest. The pages with the graphs work on the latter but not on the former. At the time of writing the report, we still have found the solution or even the root cause of this anomaly. The best theory so far is that both sets of code were created in two different environments and so therefore are not compatible with one another. This setback seriously affected the presentation of this project, as our minds had been completely perplexed by the problem.

With regards to the Angular JS code for the current rates, an Angular factory method would have been better suited to the page. This allows you to merge data from multiple API calls in to one array. This would have allowed better DOM manipulation of the data pulled in. For example, instead of having rendered today's and yesterday's rates, we could have compared both sets of data and show a graphical representation of the comparison. (A green arrow would show today's rate is higher than yesterday and a red arrow shows it is lower). This represents the only downside of Angular JS. It is very hard to find information about Angular functionality such as the Angular.factory and any information found was very hard to follow. When the documentation for Angular improves, I will definitely look forward to using it in the future.

What I have learned from the project is that, is that better preparation is required before commencing coding. For example, everyone in the team should make sure that they have the same environment is set up on each workstation before coding. I strongly believe we would have had a complete project if we had made sure of this from the start.

At present, we are unsure how to troubleshoot these issues, and continue building on this project as we intend to and it is difficult to source thorough documentation and guidance on new technologies such as Angular.

In its current presentation, this project appears unsuccessful and there are too many issues for all the various elements to work together. However, underneath, there is some very successful code and functionality and we have all learned a great deal from working on this.

References

Is Digital Currency Real Money? | americanmonetaryassociation.org. 2014. Is Digital Currency Real Money? | americanmonetaryassociation.org. [ONLINE] Available at: http://americanmonetaryassociation.org/blog/is-digital-currency-real-money/. [Accessed 05 May 2014].

Top 100 digital currencies by social media presence | Crypt.la. 2014. Top 100 digital currencies by social media presence | Crypt.la. [ONLINE] Available at:http://crypt.la/2014/03/01/top-100-digital-currencies-by-social-media-presence/. [Accessed 05 May 2014].

. 2014. . [ONLINE] Available at: https://docs.angularjs.org/guide/introduction. [Accessed 05 May 2014].

11 Reasons to Use Twitter Bootstrap. 2014. 11 Reasons to Use Twitter Bootstrap. [ONLINE] Available at: http://www.sitepoint.com/11-reasons-to-use-twitter-bootstrap/. [Accessed 05 May 2014].

What is Twitter Bootstrap? - Marketing & PR - Infragistics.com Blog . 2014.What is Twitter Bootstrap? - Marketing & PR - Infragistics.com Blog . [ONLINE] Available at:http://www.infragistics.com/community/blogs/marketing/archive/2013/04/17/what-is-twitter-bootstrap.aspx. [Accessed 05 May 2014].

Bootstrap. 2014. Bootstrap. [ONLINE] Available at:http://getbootstrap.com/2.3.2/. [Accessed 05 May 2014].

twbs/bootstrap-sass · GitHub. 2014. twbs/bootstrap-sass · GitHub. [ONLINE] Available at: https://github.com/twbs/bootstrap-sass. [Accessed 05 May 2014].

Pragmatic Rails: Write RIAs, not websites! ¶ Nicksda. 2014. Pragmatic Rails: Write RIAs, not websites! ¶ Nicksda. [ONLINE] Available at:http://nicksda.apotomo.de/2010/11/pragmatic-rails-write-rias-not-websites/. [Accessed 05 May 2014].

CSS3 Transition. 2014. CSS3 Transition. [ONLINE] Available at:http://www.w3schools.com/css/css3_transitions.asp. [Accessed 05 May 2014].

CSS3 Introduction. 2014. CSS3 Introduction. [ONLINE] Available at:http://www.w3schools.com/css/css3_intro.asp. [Accessed 05 May 2014].

The advantages of CSS3. 2014. The advantages of CSS3. [ONLINE] Available at: http://blazewebart.com/en/css/advantages-of-css3/. [Accessed 05 May 2014].

javascript - AngularJS client MVC pattern? - Stack Overflow. 2014. javascript - AngularJS client MVC pattern? - Stack Overflow. [ONLINE] Available at:http://stackoverflow.com/questions/13067607/angularjs-client-mvc-pattern. [Accessed 05 May 2014].

AngularJS on Rails 4 - Part 2 - coderberry. 2014. AngularJS on Rails 4 - Part 2 - coderberry. [ONLINE] Available at:http://coderberry.me/blog/2013/04/23/angularjs-on-rails-4-part-2/. [Accessed 05 May 2014].

RSpec.info: home. 2014. RSpec.info: home. [ONLINE] Available at:http://rspec.info/. [Accessed 05 May 2014].

Introduction to Test Driven Development (TDD). 2014. Introduction to Test Driven Development (TDD). [ONLINE] Available at:http://www.agiledata.org/essays/tdd.html. [Accessed 05 May 2014].

AngularJS Controller Tutorial. AngularJS Controller example. 2014. AngularJS Controller Tutorial. AngularJS Controller example. [ONLINE] Available at:http://viralpatel.net/blogs/angularjs-controller-tutorial/. [Accessed 05 May 2014].

Angular js, the superheroic framework for coding less and enjoy more.. 2014.Angular js, the superheroic framework for coding less and enjoy more.. [ONLINE] Available at: http://www.inqbation.com/angular-js-the-superheroic-framework-for-coding-less-and-enjoy-more/. [Accessed 05 May 2014].

MVVM Compared To MVC and MVP . 2014. MVVM Compared To MVC and MVP . [ONLINE] Available at:http://geekswithblogs.net/dlussier/archive/2009/11/21/136454.aspx. [Accessed 05 May 2014].

What is TDD, BDD & ATDD ? | Assert Selenium. 2014. What is TDD, BDD & ATDD ? | Assert Selenium. [ONLINE] Available at:http://assertselenium.com/2012/11/05/difference-between-tdd-bdd-atdd/. [Accessed 05 May 2014].

factory_girl | RubyGems.org | your community gem host. 2014. factory_girl | RubyGems.org | your community gem host. [ONLINE] Available at:https://rubygems.org/gems/factory_girl. [Accessed 05 May 2014].