New eStadium App Project Documentation

“Frontend Busters”, from eStadium webapp subteam

Spring 2015

Project goal

To have a new eStadium app that

1. looks mordern: The existing eStadium app was developed years ago, and does not fit what is currently trending on the Internet in 2015.
2. is responsive: The exisiting eStadium app has two different apps for desktop and mobile phones. Although each one is optimized for each screen size, they feel different (e.g. different looks for buttons, etc). We want to have an app that is still optimized for both screens, but feel the same. At the meantime, we want to have only one app for both desktop and mobile phones.
3. is easy for development and maintainance: This argument follows the goal of having one coherent responsive app. Since we want to have one app for both desktop and mobile phones, we will also have one codebase. That means during development and maintainence, we can develop/fix once, and test/deploy everywhere. We also want to leverage modern web technologies. An example is two-way data binding, instead of manually generating HTML.
4. can be ported to mobile devices: We also want to have native apps on mobile platforms. This would broaden the usage of our app, since people don’t have to explicitly open up a web browser to use eStadium.

Why Ionic

Ionic framework (http://ionicframework.com/):

1. looks modern: Seeing is believing. Here are some showcase (http://showcase.ionicframework.com/), and you can also look at the new app we developed.
2. is responsive: Although Ionic is designed for a mobile experiece, we can still manipulate CSS to make a different style for desktop. However the core logic part of the code remains the same, and we can make sure that we share more than 95% of the code on desktop and mobile.
3. is easy for development and maintainence: By having a responsive design using Ionic, sharing code is guaranteed. Ionic also uses Angular.js (https://angularjs.org/), which includes lots of easy-to-use modern web technologies such as two-way data-binding. Additionally Ionic also have useful command line tools. For example you can easily use “ionic serve” in your terminal from project root directory to host the app on your local machine for

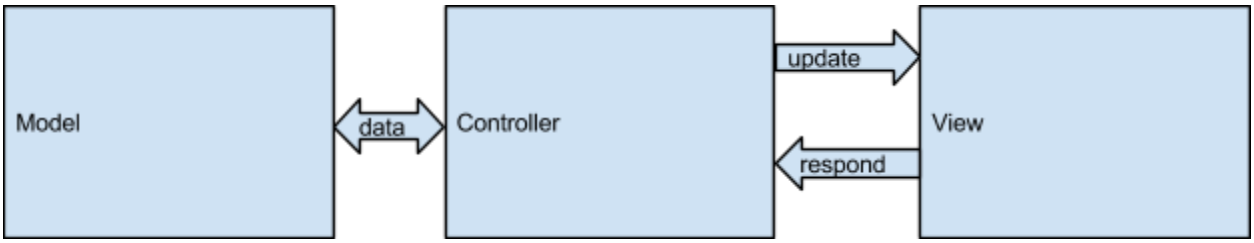
testing. It also detects changes so that you don't even to have to re-host the app to see changes. It just changes on the fly.

4. can be ported to mobile devices: You can simply use "ionic build android" or "ionic build ios" from project root directory to generate Android/iOS native apps. Not a line of Java/Swift is required. The native app is essentially a web browser that hosts our app, but it is still a native app. It has an icon on your launcher, and it is capable of accessing native platform services such as notifications (although we are not using any yet).

Overall architecture

MVC & singleton

If you are already familiar with those two nouns, and have a brief idea about them, skip this subsection.
MVC is Model-View-Controller, a software design pattern. Its main purpose is to facilitate separation of concern. Here is my understading of MVC in our



context:
Model is underlying data, such as all the plays in this game.
View is the user interface you see, such as the list of all plays in this game that you see on the play by play page.
Controller is the glue between models and views. It retrieves data from models, and populates the view with the data. The reverse direction of actions can also happen, but it's more rare in our app.

Singleton is easy to understand. It is the one single module in the software that can be accessed from anywhere.

MVC & singleton in Angular.js

If you had experiences with Angular.js, you can probably skip this subsection.
In Angular.js:

- Models (aka data) can come from various soruces. They can come from
  - Hardcoded data
  - Local files
  - Remote servers (this is what we are aiming for)
- Controllers are controllers, as defined in MVC.
- Views are templates. We fill templates with data, using controllers.

- Controllers and views are glued with routing.
- Singletons are services.

Implementation detail

What is each folder and file?

So you have a codebase that you are not familar with, the very first thing you want to do is to run it. Once you have cloned the repository, use your system terminal to run “ionic serve” (if you don’t have Ionic command-line tools installed, you should) under the project root. The project root is html/new (NOT html/new/www). Once you run this command, your default browser should pop up to show the app. However, since you are running the app in your local browser, and this app is making network calls to the remote eStadium server, your browser blocks this call due to security reasons. To remove the blockage, install this extension in your Chrome (search for similar tools if you are using other browsers):
https://chrome.google.com/webstore/detail/allow-control-allow-origi/nlfbmbojpeacfghkpbjhddihlkkiljbi.
Once you have installed it, a red “CORS” icon should show up on right top corner of Chrome. Click on that, turn on that big switch, and refresh the page. You should see the app working then.

So now you have the app running, what is inside?
Everything that matters in under www/ folder
Under www/ folder:

- index.html: The entrance point. When the app loads, this is the first file that your browser reads. You probably don’t need to modify this file, unless you want to add/remove Javascript files. If you need to do that, you are probably very familiar with the codebase.
- js/app.js: The entrant Javascript file. It is loaded by your browser, right after it has loaded all libraries dependencies (Angular/Ionic), but before all your other Javascipt files are loaded. If you need to add new items on the left drawer (mostly probably also add a new page), you will need to modify this (along with other files that will be mentioned later).
- js/controllers.js: This is where all controllers reside. When you are making a new page, you will need to add a new controller to control this page. Gluing the newly added controller with a new view will be discussed later.
- js/services.js: This is the “service” (aka singleton) where the models (aka data) are retrieved via network calls to remote

eStadium server, and stored. You will need to add new functions to this service if you are making a new type of network call.

- js/object_utils.js: This is another “service” (aka singleton) that includes a bunch of utilities.
- templates/menu.html: This is the skeleton of the app that you actually see on your browser. By skeleton I mean the top yellow bars, the left drawer, and the main content area. This file does not include anything that relates to content of each individual page though. By the way this is a view, and it is controlled by AppCtrl (app controller) in js/controllers.js. How that gluing is done will be discussed later. If you need to add new items on the left drawer (mostly probably also add a new page), you will need to modify this.
- other files under templates/: Those are, again, views. They are called templates. Your controllers are tied to those templates. When each inidiidual page loads, controllers retrieve data and fill those templates with data. If you need to add new items on the left drawer (mostly probably also add a new page), you will need to add a new template here.
- css/: Styles. Those enable us to have a different style for desktop (e.g. bigger font sizes) while keeping everything mentioned above intact.
- img/: Images, should be self-explanatory.
- lib/: Angular/Ionic libraries files, don’t touch unless you are updating Angular/Ionic version. If you need to update Angular/Ionic version, be aware that the Ionic CSS style was modified to make video player full width/height.

Index & routing & controllers & templates

js/app.js contains configurations for app “states”. Interpret the word “states” as different pages in our context. Here is a list of events that happen when you click on “Homepage” on the left drawer:

1. When you click on “Homepage”, the “href” is retrieved, from templates/menu.html. In this case the href is “#/app/homepage”
2. This href is sent to js/app.js. Note you have only declared configurations in app.js, but not the actual implementation. This href is handed to Angular, and Angular looks up what page should it display, according to

your configurations. In this case, one of our configurations is this:

```
.state('app.homepage', {
  url: "/homepage",
  views: {
    'menuContent': {
      templateUrl: "templates/homepage.html",
      controller: "HomepageCtrl"
    }
  }
})
```

3. Angular can match the declared “/homepage” with the href retrieved “#/app/homepage”. Note the href “#/app/homepage” will be automatically curtailed to “/homepage” by Angular since we have declared an “abstract” route “/app” above.
4. Angular then knows that it should display the homepage in the main content area, and it does this by looking up the above configuration. It will find that the template (aka view) would be templates/homepage.html, and the controller be HomepageCtrl.
5. Angular will then look up for templates/homepage.html.
6. Angular will then find a controller that is called HomepageCtrl. It can find

```
.controller('HomepageCtrl', ['$scope', '$ionicModal' ,'eStadium', 'host', function($scope, $ionicModal, eStadium, ho
```

   it because we have declared it in app/controllers.js!
7. After running the code that is under this controller, data will be retrieved and populated in the template. Everything completes.

From those steps, you probably know that if you want to add a new page, you would have to:

- Add a new item in the left drawer so that a user can click on it to get to the new page. This happens in templates/menu.html. This new item will also have a href.
- Add a new configuration in app.js, with “url” matching later part of the above mentioned href. The very first field (“app.homepage” in above code snippet) is the name of the state. Make something meaningful of it, but it is not referred anywhere in our case.
- Add a new template under templates/ folder
- Add a new controller in controllers.js.
- Glue the new template and the new controller in the new state you created.

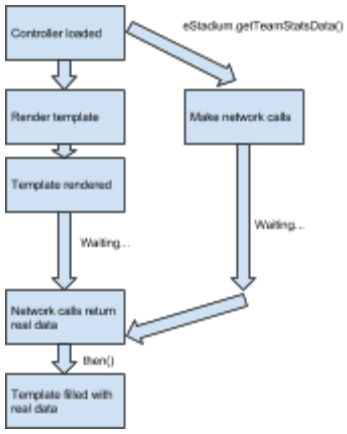
The eStadium service

Asynchornous networking calls

Although people have deveolped many advanced network technologies such as Wi-Fi and 4G/LTE, network operations, such as getting all plays

of the current game from the remote eStadium server, is still 1000 times slower than how fast CPU/GPU can render the list of all plays on your screen. Thus we don't want to hang up on the users while we are waiting for the network calls to finish. This makes all networking calls "asynchronous", meaning those calls do not follow the steps of CPU/GPU rendering process. Those calls can finish at arbitary time in the future, although the elapsed time is often trivial from a user's perspective, thanks to those advanced network technologies.

Continuing with the previous subsection, when a controller (here I am taking TeamStatsCtrl as an example) is loaded, a typical controller in our app will follow the following events



At the beginning of the controller code (ignoring the giant render() fucntion that wraps the code for now), it calls eStadium.getTeamStatsData(). This function does not directly return the real data, as you might expected. It instead returns a promise, which we will discuss later. Later the then() function will be called, when the network operation returns real data from the eStadium server. It does not happen immediately from the perspective of rendering, but some time later in the future. When the then() function is called, it will also give the real data returned from server in the "data" parameter passed in. You can then utilize this data to populate the template.

Promise

Now we know the API exposed by the eStadium service. Every get***() function, if you look at all the controllers, does not return the real data immediately. Those function calls all return "promises".

A promise is a promise. If I owe you 10 bucks, I might give you a promise that "I will pay you back a week later". Hopefully I will fulfill my promise, not now, but a week later. The whole eStadium service is just

a collection of such money borrowers. Calling eStadium.getTeamStatsData() will not pay back the money (data), but the promise to pay back the money (data) in the future.

Look at the code for eStadium.getAllDrives():

```
factory.getAllDrives = function () {
    var deferred = $q.defer();

    if (all_drives_data != null) {
        deferred.resolve(all_drives_data);
    } else {
        factory.getGlobalData()
            .then(retrieve_global_data)
            .then(retrieve_all_drives_data)
            .catch(error_out);
    }

    function retrieve_global_data(global_data) {
        return $http.get(
            host + '/service/get/drivetracker/getDrivesAll.php',
            {params:{"gameid": global_data.game_id}}
        );
    }

    function retrieve_all_drives_data(result) {
        var data = result.data;
        all_drives_data = {
            "global_data": global_data,
            "data": data.drives
        };
        deferred.resolve(all_drives_data);
    }

    function error_out(data, status) {
        deferred.reject(status);
    }

    return deferred.promise;
};
```

The first line contructs a promise called “deferred” (it’s a deferred promise).

The following 3 lines are for caching, which will be discussed later. But focus starting with factory.getGlobalData().

“factory” is what is used to build this service. We are calling another function “getGlobalData()” inside this service. This function “getGlobalData()” also returns a promise, which will give you all the

global data of the current game that is diaplaying in the app. Examples of global data include: game id to make other network calls because most of the API's require this, team names, paths of team logos, etc.

Then you see the familiar "then()" call. Inside is a function, which is declared lines below.

Inside this retrieve_global_data() function, global_data parameter is used to access the game id, in order to make another network call to get all the drives data. This function returns yet another promise, but this promise is hidden by "$http.get()". Google this, and you will find out how to use this function. It is actually a part of Angular.js framework, and it indeed returns a promise.

Going back to the "then()" call, it is chained with yet another "then()" call. Inside is yet another functions, which is yet declared lines below. Sounds familar? Yes, we are essentially chaining promises! Once the factory.getGlobalData() promise is resovled, we will make another network call in retrieve_global_data. Once the promise for the network call in retrieve_global_data is resolved, we will call retrieve_all_drives_data.

What is inside retrieve_all_drvies_data? It should get all the drives data first of all. But what's more important is to return that data to whoever is calling this function! Remember our "deferred"? It's finally the time to fulfill this promise. We use deferred.resolve() to fulfill the promise. Remember what should happen when you have fullfilled the promise? The outside then() of the caller will be called. In this case the caller, DriveTrackerCtrl's then() will be called, which gets the data and fills the template for drive tracker.

Finally at the end of this getAllDrives(), we return the promise by returning deferred.promise. It's not the real data yet, it's the promise for the real data.

Caching

Network calls are expensive. They are battery, bandwidth and network traffic killers. We want to avoid them as much as possible. In our context, when we are at the homepage, then navigate to drive tracker page, and then navigate back to homepage, we don't want to make yet another network call to get data for homepage, unless you decide to manually refresh to get updates (it then becomes user's own liability of battery, bandwidth and network traffic).

This is why we cache data, unless user decides to manually refresh, or to switch to see another history game.

Again look at code for eStadium.getAllDrives():

```
factory.getAllDrives = function () {
    var deferred = $q.defer();

    if (all_drives_data != null) {
        deferred.resolve(all_drives_data);
    } else {
        factory.getGlobalData()
            .then(retrieve_global_data)
            .then(retrieve_all_drives_data)
            .catch(error_out);
    }

    function retrieve_global_data(global_data) {
        return $http.get(
            host + '/service/get/drivetracker/getDrivesAll.php',
            {params:{"gameid": global_data.game_id}}
        );
    }

    function retrieve_all_drives_data(result) {
        var data = result.data;
        all_drives_data = {
            "global_data": global_data,
            "data": data.drives
        };
        deferred.resolve(all_drives_data);
    }

    function error_out(data, status) {
        deferred.reject(status);
    }

    return deferred.promise;
};
```

On second line that we skipped on previous section. What is this all_drives_data? Well, it is a global variable, defined in this service:

```
var all_drives_data = null;
```

It is initialized to null. When you first use this function getAllDrives(), all_drives_data is null, so it will go ahead and do the chained promise (network calls) we discussed easiler.
When it comes to retrieve_all_drives_data(), this all_drives_data is assigned with the real data! Note a convention here. Since mostly we

are calling one function in eStadium service within one controller, the data being returned also includes the global data. So if you want to use team name in drive tracker page, you can.

Imagine the second time this getAllDrives() function is called. all_drives_data is not null! So you would just resolve the promise, with this exact data, without making any network calls.

Refresh & Time-machine

Again, think of the previous section of how caching works. A controller will call a function in eStadium service. This service function will find out the data is cached, so it just returns (or I shall say resolves) the cached data. This is not the data you want to get when you are refreshing and trying to get new data from the server. So what we want to do is to

1. Invalidate all the cached data
2. Re-render the view

Those two steps are what we exactally do.

Check out AppCtrl, which controls the refresh button. When the refresh button is clicked, $scope.refreshClicked() will be called. Inside this function, eStadium.nullifyAllData() is called. This function essentially reset all cached data back to null so that when you re-render the view, you will retrieve new data from the server. $scope.refreshAll() is then called, which is essentially another function with AppCtrl. Inside $scope.refreshAll(), it calls built-in Angular function $scope.$broadcast. Broadcasting is one of the ways that a controller can talk to other controllers. In this case, AppCtrl broadcasts a “refresh” message. If you look at all other controllers, they all have this $scope.$on function. Those $scope.$on functions ensure that when other controllers broadcast a “refresh” message, the controllers can re-render their view. This is indeed what those $scoop.$on functions do. They all have the render() function, which essetially renders (or re-renders) the view.

Discussion on time-machine will be a little more invlovled, but suffient code comments are left there so that you can figure it out. It’s basically a special case of refreshing: invalidating all caches and re-render the view, but using a different game id to get the data.

Notice for refresh/time-machine to work, a hack is used. In app.js, caching views are disallowed: $ionicConfigProvider.views.maxCache(0). Ionic by default also caches views (templates that are already filled with real data) so that if you have the same route (e.g. getting to homepage by “/homepage”), you don’t

have to re-render. But this is against our idea of re-rendering views when you want to refresh.

Bugs & Improvement

See TODO.md under the project root.

Tips & Tricks

1. Google is the first friend that you should think of when you have questions.
2. Use an IDE. It is so much less error-prone, and more productive. You really don't want to waste time on a plain old text editor to figure out that you are missing a "}" in your Javascript. Unless you believe you are a super-coder (I mean Linus Torvalds type), please use an IDE. Sublime/Atom would still suffice and prevent you from missing a "}", but a proper IDE is still recommended. WebStorm is our option, and it has emacs/vim plugins and Sublime/Atom color schemes if you really miss them. It is a commercial product, but you can get it free if you have an edu email address (we have @gatech.edu).
3. Don't use the GitHub desktop app. Use either terminal on your system, or WebStorm's built-in Git functionality if you are using WebStorm.
4. Always pull from Git repository before you proceed coding, everytime.
5. If you are refering to a file in the project: For example if you want to refer to an image "test.png" that is resided under img/ folder, use "img/test.png". Using other formats might work if you are running under browser, but not work on native mobile apps since they are more strict on paths.