Gr8Conf 2012 Workshop Guide

The purpose of this document is to provide a brief description of each Javascript library we will use and to guide the reader through the process of building or sample app, twitterMonitor.

Structure - Backbone.js (and Underscore.js)

Backbone.js provides exactly what it sounds like: a solid foundation from which to build a large Javascript based web app. It provides structure and functionality for Models and their corresponding Views, Collections, History, and a custom Event system. It allows a developer to quickly set up synchronization between a model and the server. It does not, however, provide much in the way of Controller structure, and it is thus up to the developer to handle that portion of the application. Backbone has two requirements: Underscore.js (a functional 'utility belt' which Backbone takes advantage of) and either jQuery or Zepto.

We encourage you to read through the Backbone documentation (or at least have it open as a reference) during this project, but we highlight here a few important bits to keep in mind. In addition, the un-minified version of the Backbone source is highly readable; we recommend looking through it.

First off, when creating a new Backbone object, one 'extends' a from a base type, like so:

```
TM.Models.Keyword = new Backbone.Model.extend({options});
```

The extend method accepts an options parameter, which is a normal JS object that contains additional functionality. A base Backbone object inherits quite a bit of functionality, but the options parameter can override anything you wish. For example, each Backbone object contains a constructor method called 'initialize' which is empty by default.

Below are listed some important features and methods from each object type

Models

Models typically act as a direct mapping from an object on your server to one in your UI. Each Backbone model knows how to access its mapped Model (by default via a pure REST implementation) and allows a range of functionality, including performing client-side validation of objects.

fetch({options})

The fetch method pulls information from the server and updates the model with any new data. If data has been changed, a 'changed' event is fired from this object. The options parameter allows for 'success' and 'error' callback methods.

Views

Just like you'd imagine, Views are responsible for rendering and code surrounding the DOM nodes they've created. When creating a new view, one typically attaches the model, then calls render() followed by bindEvents() (or use a declarative events object). One nice feature is that Backbone encourages the use of JS Templates.

render()

Responsible for rendering some set of nodes, typically based on a model. If done nicely, a typical render method may look something like this:

```
render: function () {

    this.$el.html($(TM.Templates.tweet({
        text: this.model.get("text")
    })));
    return this;
},
```

Note the 'this.$el' object, which is a cached jQuery/Zepto selector for the view's root node, and the use of 'return this'. A view is *not*, by convention, responsible for inserting itself in the DOM; that task generally falls to whatever created the view.

Collections

Collections provide a range of convenience methods for managing ordered sets of Models. One of the most interesting features is the ability to hook into a 'list' type action on the server to auto-instantiate a set of Models.

Templating - Handlebars

Based on the Mustache template format, Handlebars is an excellent choice for building Javascript templates. Usage is simple: create a set of Templates (which contain named placeholders for data) and compile them, which transforms them into a Javascript function. This function accepts a context - an object containing data mapped to your named placeholders - and returns a string which one can inject into your page as a DOM element.

Templates can placed in an HTML page as a node that your scripts must locate, or included as escaped strings in your Javascript.

Testing - Jasmine

Jasmine is a BDD testing framework for Javascript. The library creates a highly readable DSL for describing your tests, which are referred to as 'specs'.

About this App

The included Grails application, called 'twitterMonitor', is intended as an introduction in how to build responsive, API-driven, Javascript-heavy applications. The overall purpose of this sample application is allow a user to monitor Twitter for a set of keywords, and their occurances within Tweets. The number of occurences are tallied and displayed to the user, as well as the Tweets.

While all code is bundled within one Grails application, it is helpful to think of this as two separate applications, the client-side and the server-side. The server-side component is largely already complete, and this guide instead focuses on how to rebuild the client-side code. Below are high-level descriptions of each.

Server

The twitterMonitor server will receive data from the front-end, either a string representing a keyword to add or a number representing the id of a keyword to delete. It will also send data about the keywords and saved tweets to any client in JSON format.

The server also periodically executes two Quartz jobs: one which deletes tweets older than a specific threshold, and another which searches Twitter for new tweets containing the keywords. When a keyword is first searched, twitterMonitor will grab the previous n tweets (where n is a config option), but subsequent searches will return new tweets in between the last seen tweet. Keywords with matches will have a counter increased, and matching tweets are saved, then subsets of data are sent to the client.

Client

The twitterMonitor client, once complete, will display information about keywords and tweets. The UI will contain three major visual components: a 'control' area (allows a user to input new keywords and start/stop the Client to server communication), the keyword area (displays each keyword with number of occurences and a bar graph expressing relative counts), and the tweet queue (displays tweets containing the keywords).

File Locations

Nearly every file we add or edit today will be within web-app/js/src/. Please note that if the reader changes any file names or adds new files, the file must be added to the ApplicationResources.groovy in grails-app/conf so that the Resources plugin will bundle it for you.

Instructions

The following guide will walk you through the steps needed to build the twitterMonitor UI; please follow it at your own pace. A few notes:

1. Try to keep Javascript objects and CSS class names the same as the guide; otherwise you'll need to change the corresponding values in multiple places.
2. This guide assumes that the reader has created a Grails app before, and is aware of the standard file locations and commands.
3. Our apologies ahead of time for any bugs that may have crept in.
4. Refer to the Backbone.js, Handlebars, or Jasmine docs often for further clarification.
5. Be Creative! This document is just a guide; there's much more that could be done with the information here. For example, the underlying service could be updated to capture much more information about each tweet, which could lead to more in-depth UIs. Also, the Tweet queue intentionally does not use a Collection; one could edit it to make use of the Collection object.

Getting Started

I'll assume that you 1) have the full twitterMonitor project (e.g. downladed the repo from github) and 2) you have grails version 2.0.3 installed. Great. Now:

1. See if there's anyone in the room without a computer, who may be looking around nervously. Buddy up with them and offer to pair program!
2. Start the application with 'grails run-app'

At this point, navigate to http://localhost:8080/twitterMonitor; you should see a blue banner with the words 'Twitter Monitor' and a blank white screen.

Let's get started!

1. Models

We will need to create two models: Keyword and Tweet.

- Locate and open the the file web-app/js/src/models/tweet.js
- Create the model and add it to our TwitterMonitor JS namespace by typing:

```
TM.Models.Tweet = Backbone.Model.extend({});
```

Congratulations, you've created your first Backbone Model! We could certainly make this more complicated by adding validation, defaults, etc, but it's not necessary for this demonstration. We've successfully extended the default BackboneModel into our own Tweet Model, and namespaced it into TM.Models (see web-app/src/js/core/base.js to see how the app is namespaced).

- Locate and open web-app/js/src/models/keyword.js
- Add the following:

```
TM.Models.Keyword = Backbone.Model.extend({
    url: function () {
        return "/twitterMonitor/keyword/" + this.get("id");
    }
});
```

We've just created our second Model, this time extendeding it's base functionality by defining a url function. Backbone will use an Object's url function or object (you can use either) to know how to uniquely fetch this model's data; it should map to our keyword's endpoint on the server. We're not quite finished with the model yet; let's add a function to help calculate the relative size of the keywords hit count. The function should return a ratio of its 'numSeen' variable to that of the other keywords. We can take advantage of this by using the model's collection parameter.

- Enter the following (or something similar) into the Model's extend object:

```
getBarPercentage: function () {
    var maxSeen = this.collection.getMaxNumSeen(),
        percentage = (this.get("numSeen") / maxSeen) * 100;
    // ensure a max percentage of 100
    if (percentage > 100) {
        percentage = 100;
    }
    return Math.round( percentage );
}
```

In the code, 'this' refers to the model itself (you may note that throughout we often cache 'this' into a local variable named 'self' in the cases where 'this' will be of an improper scope). The 'collection' parameter is a reference to a collection object that the model is a member of; we'll create the 'getMaxNumSeen' function later on, when we create the Collection.

- Let's set a default value for 'numSeen'. Add the following to the Model's extend object:

```
defaults: {
    "numSeen": 0
}
```

Make sure to separate each of our objects/functions within the extend object with commas! In the end, you should have something like this:

```
TM.Models.Keyword = Backbone.Model.extend({
    defaults: {
        "numSeen": 0
```

```
    },

    url: function () {
        return "/twitterMonitor/keyword/" + this.get("id");
    },

    // Returns the relative percentage of the bar graph's width compared with the other models in the collection
    getBarPercentage: function () {
        // A model that becomes part of a collection gets assigned a reference
        var maxSeen = this.collection.getMaxNumSeen(),
            percentage = (this.get("numSeen") / maxSeen) * 100;
        // ensure a max percentage of 100
        if (percentage > 100) {
            percentage = 100;
        }
        return Math.round( percentage );
    }

});
```

2. Collections

Seeing as how we reference the Collection in the previous section, let's work on that next. Collections are a great way for managing groups of Models; they provide Array-esque methods for adding/removing, events for when items are altered, and (my favorite) access our keyword/list end point to automatically instantiate our set of keywords.

- Open up web-app/src/js/collections/keywords.js and enter the following:

```
TM.Collections.Keywords = Backbone.Collection.extend({
    url: "/twitterMonitor/keyword", // maps to the 'list' action on our server

    model: TM.Models.Keyword,

    // in addition, this collection also keeps track of the current maximum 'numSeen' of the models in its care
    // this maximum is used to help set the width of the bar graphs on each keywords' view. The width of the bar graph
    // is relational to the keywords' current count versus the others (i.e. the bar graph will have a max width of 100%)
    //
    // Thus, we default to 100
    defaultMax: 100,
    maxNumSeen: 0,

    initialize: function () {
        this.maxNumSeen = this.defaultMax;
    }
});
```

We do not use the 'defaults' object here to help highlight the constructor function 'initialize'. This collection, beyond simply holding and syncing the Keywords, should also be aware of the maximum 'numSeen' value in its keywords, so that each keyword can know its proper relative width that its View will display. The Collection's maxNumSeen value should return either maxNumSeen or the defaultMax (100), whichever is greater.

- Add the following to your extend options object:

```
// returns maxNumSeen or defaultMax, of maxNumSeen has not been set yet
getMaxNumSeen: function () {
    return this.maxNumSeen ? this.maxNumSeen : this.defaultMax;
}
```

But now we need a function that will discover the maxNumSeen for us. Luckily, we can make

use of the Collection's 'pluck' function.

- Add the following function:

```
findMax: function () {
    // the 'pluck' function extracts a value from each model in this collection, and places those values in an Array
    var counts = this.pluck("numSeen"),
        i = counts.length,
        max = 0;

    while (i--) {
        if (counts[i] > max) {
            max = counts[i];
        }
    }
    this.maxNumSeen = max > this.defaultMax ? max : this.defaultMax;
}
```

That's nice, but when should we search the models for the maximum? Whenever we call getMaxNumSeen? We could... but that's no fun. Let's instead hook into the Collection's event system! Whenever a member Model is added, removed, or updated, the Collection fires events that we can listen for. Where should we listen for these events? There is certainly an argument to be made that we should add the bindings during the initialize function. However, Backbone Views have a convention where the rendering of an item and the binding of its events are kept as separate steps. Let's extend that notion to our Collection.

- Add the following function to the extend options object:

```
bindEvents: function (){
    var self = this;
    // "reset" is fired when the collection is created or reloaded
    this.on("reset", function (){
        self.findMax();
    });
    // "change is fired when a value in the collection is altered or a new item is added"
    this.on("change", function () {
        self.findMax();
    });
    // "destroy" is fired when a model is removed
    this.on("destroy", function () {
        if (self.models.length === 0) {
            // alert the higher-ups that the collection is empty
            self.trigger("empty");
        }
    })
}
```

In the end, your collection should look something like:

```
TM.Collections.Keywords = Backbone.Collection.extend({
    url: "/twitterMonitor/keyword",

    model: TM.Models.Keyword,
    // in addition, this collection also keeps track of the current maximum 'numSeen' of the models in its care
    // this maximum is used to help set the width of the bar graphs on each keywords' view. The width of the bar graph
    // is relational to the keywords' current count versus the others (i.e. the bar graph will have a max width of 100%)
    //
    // Thus, we default to 100
    defaultMax: 100,
    maxNumSeen: 0,

    initialize: function () {
        this.maxNumSeen = this.defaultMax;
    },

    bindEvents: function (){
```

```
        var self = this;
        this.on("reset", function (){
            self.findMax();
        });

        this.on("change", function () {
            self.findMax();
        });

        this.on("destroy", function () {
            if (self.models.length === 0) {
                // alert the higher-ups that the collection is empty
                self.trigger("empty");
            }
        })
    },

    // looks through the collection for the maximum numSeen value
    findMax: function () {
        // the 'pluck' function extracts a value from each model in this collection, and places those values in an Array
        var counts = this.pluck("numSeen"),
            i = counts.length,
            max = 0;

        while (i--) {
            if (counts[i] > max) {
                max = counts[i];
            }
        }
        this.maxNumSeen = max > this.defaultMax ? max : this.defaultMax;
    },

    // returns maxNumSeen or defaultMax, of maxNumSeen has not been set yet
    getMaxNumSeen: function () {
        return this.maxNumSeen ? this.maxNumSeen : this.defaultMax;
    }
});
```

We intentionally do not create a Collection for the Tweet model, in order to demonstrate their convenience when we work with Tweets later on. We leave to you the task of working in a Tweet Collection.

3. Templates

Before we move on to the Views, let's take a look at Handlebars Templating. We've created much of the templating already, which you can find stored currently as a partial located at grails-app/views/standAlone/_handlebars.gsp. This partial is rendered on a page that will display our app and adds a series of script blocks (with the type 'text/x-handlebars-template'). Each of these blocks are our individual templates; our client-side code will use these templates as the building blocks of the UI. The client-side code will store each of these templates by their id attribute as a function in the namespaced object 'TM.Templates'.

If you're curious how this works, examine the 'compileTemplates' function located in 'web-app/js/src/core/utils.js'. In order to use a Handlebars template, you must 'compile' it. To avoid doing this step whenever we need to use template, this function 'pre-compiles' each of the templates on our page by looking for script blocks with the 'text/x-handlebars-template' type.

Let's move onto Views in order to see how these Templates are used in practice.

4. Views

We have two types of Views in twitterMonitor: 1) those that are directly responsible for rendering and managing a Model, and 2) those that are responsible for rendering and managing sections of the page. In that sense, Backbone Views are analagous to Controllers (with the Handlebars Templates being our 'views'), although we do include logic in our Backbone Views that manipulates the DOM directly. In a larger project, it may be wise to pull this functionality out into separate objects.

Model Views

Let's first tackle the Keyword view:

- Open web-app/js/src/views/keyword.js
- Add the following:

```
TM.Views.Keyword = Backbone.View.extend({

    initialize: function (options) {
        //attach a reference on the model so that interval driver knows to delete this view without having to search for it
        this.model.attachedView = this;
    },

    render: function () {
        var self = this;
        $(this.el).html(TM.Templates.keyword({
            text: this.model.get("text"),
            numSeen: this.model.get("numSeen")
        }));
        this.updateGraphWidth(); //we'll add this next
        return this;
    },

    bindEvents: function () {
        var self = this;
        // add a click handler to remove the keyword
        this.$el.find(".keyword-remove").on("click", function () {
            self.destroy.call(self);
        });

    }
});
```

When a View is created, a model is passed to it (as we'll see later), which is then attached and referenced as 'this.model'. A view contains a reference to it's DOM node in the form of 'this.el' and 'this.$el' (jQuery/Zepto cached version of this.el), which we use here in the 'render' function. Note the use of the template, and the fact we pass in an object of values from the model (compare the names of the object with the values in its Template). Finally, note the fact that render returns the view itself. By convention, a View does not insert itself in the DOM, it's up to the object that creates the View to determine when to do the insertion.

One convention is to bind a 'change' event on the view's model and re-render when a change occurs. That would work just fine here, but we're added some fun css animations for when

the bar graph should change. To accomplish, let's add functions to update the bar graph width and the counter display.

- Add the following to the extend object:

```
// convenience method to handle these two functions as a single callback
updateDisplayValues: function () {
    this.updateGraphWidth();
    this.updateDisplayCount();
},

// update the bar graph width based on the model's
updateGraphWidth: function () {
    this.$el.find(".bar").width(this.model.getBarPercentage() +"%");
},

updateDisplayCount: function () {
    var self = this;
    self.$el.find("span.num-seen").text(self.model.get("numSeen"));
}
```

The bindEvents method adds some functionality to destroy the View, which is a default Backbone function. However, we should also destroy the Model, and only do so after wev'e alerted the server to our change. This can be accomplished by overriding the default destroy fuction.

- Add the following to the extend object:

```
// responsible for deleting the keyword on the server and destroying this view
destroy: function () {
    var self = this;
    //attempt to delete from the server, if successful we proceed with UI removal
    self.model.destroy({success: function () {
        self.removeUI.call(self);
    }});
},

// fancy removal
removeUI: function () {
    var self = this;
    self.$el.unbind(); //clear any bindings
    self.$el.fadeOut("slow", function () {
        //remove view from the dom
        self.remove();
    });
}
```

The keyword Model's destroy function will, by default, make a DELETE call to /twitterMonitor/keyword/. It accepts a 'success' callback that is fired if the delete was successful on the server's end.

Removing a View in Backbone is accomplished in a few steps: 1) unbind all event listeners and 2) call the view's remove() function.

At this point, our Keyword View should look like:

```
TM.Views.Keyword = Backbone.View.extend({

    initialize: function (options) {
        //attach a reference on the model so that interval driver knows to delete this view without having to search for it
        this.model.attachedView = this;
    },
```

```
    render: function () {
        var self = this;
        $(this.el).html(TM.Templates.keyword({
            text: this.model.get("text"),
            numSeen: this.model.get("numSeen")
        }));
        this.updateGraphWidth();
        return this;
    },

    bindEvents: function () {
        var self = this;

        this.$el.find(".keyword-remove").on("click", function () {
            self.destroy.call(self);
        });

    },

    // convenience method to handle these two functions as a single callback
    updateDisplayValues: function () {
        this.updateGraphWidth();
        this.updateDisplayCount();
    },

    // update the bar graph width based on the model's
    updateGraphWidth: function () {
        this.$el.find(".bar").width(this.model.getBarPercentage() +"%");
    },

    updateDisplayCount: function () {
        var self = this;
        self.$el.find("span.num-seen").text(self.model.get("numSeen"));
    },

    // responsible for deleting the keyword on the server and destroying this view
    destroy: function () {
        var self = this;
        //attempt to delete from the server, if successful we proceed with UI removal
        self.model.destroy({success: function () {
            self.removeUI.call(self);
        }});
    },

    // fancy removal
    removeUI: function () {
        var self = this;
        self.$el.unbind(); //clear any bindings
        self.$el.fadeOut("slow", function () {
            //remove view from the dom
            self.remove();
        });
    }
});
```

Next, let's create the individual Tweet View.

- Open web-app/js/src/views/tweet.js
- Add the following:

```
TM.Views.Tweet = Backbone.View.extend({

    render: function () {
        var ctx = this.mapModelToContext();
        this.el = $(TM.Templates.tweet(ctx));
        return this
    },

    // because generating the context for rendering has some small variability,
    // we pull out the generation to a separate function for easier testing
    mapModelToContext: function () {
        return {
            class: this.model.id % 2 === 0 ? "even" : "odd",
            imageUrl: this.model.get("profileImageUrl"),
            text: this.model.get("text"),
            userName: this.model.get("userName")
```

```
        };
    }
});
```

Not much to it; the only abnormal bit is using the model's id to determine whether to add an 'even' or 'odd' css class (take a look at the corresponding Template for reference, if you'd like).

Container Views

On to the container Views! These will manage the Keyword and Tweet sections of the page.

- Open web-app/js/src/views/keyword-container.js
- Add the following:

```
TM.Views.KeywordContainer = Backbone.View.extend({

    initialize: function () {
        this.keywords = new TM.Collections.Keywords(); //instantiate the collection
        this.keywords.bindEvents(); // bind its events!
        this.views = []; // set up an array to track our created views
    },

    render: function () {

        $(this.el).html(TM.Templates.keywordContainer({}));
        return this;
    }
});
```

Nothing fancy here, we instantiate the keyword collection, bind its events, and setup the rendering function

- Next, let's create function to kick off the collection's fetch mechanism. Add the following to the extend object:

```
reloadKeywords: function (add) {
    var self = this;
    this.keywords.fetch({
        add: add,
        success: function (collection, data) {
            self.populateKeywords.call(self, collection, data);
        }
    });
}
```

Here we trigger the fetch command on the collection; as stated before this will automatically create the necessary models. Convenient! In addition, we pass a success parameter which triggers a 'populateKeywords' function, which we'll get to next. The 'add' option is of note as well: by default, a Collection fetch will recreate the entire collection, wiping out models and starting over, which is a bit overkill for our purposes. Instead, one can use the 'add' option, which if true will add new items to the collection without resetting the whole thing. The

disadvantage of course is that we do not 'prune' deleted keywords.

- Add the 'populateKeywords' function:

```
populateKeywords: function (collection, data) {
    var self = this;

    if(self.views.length === 0) {
        this.$el.html("");
    }
    //underscore.js's 'each' iterator function
    _.each(collection.models, function (model) {
        self.createView.call(self, model);
    });

    //display empty message
    if (collection.models.length === 0){
        this.showEmptyMessage()
    }
}
```

The function accepts the built collection and a data attribute, passed in from the success callback, although we only use the collection.

- Let's add the two functions referred to above:

```
showEmptyMessage: function () {
    this.$el.html(TM.Templates.keywordContainerEmpty());
},

createView: function(model) {
    //first, ensure the view hasn't already been created

    if (!model.attachedView) {  // attachedView is set on the model's initialize
        var view = new TM.Views.Keyword({model:model});
        //render it initially
        this.$el.append($(view.render().el));
        //set the element on the new keyword
        view.setElement(this.$el.children().last());
        //and bind!
        view.bindEvents();
        //and store. We'll need to access the view object's reference later for destruction
        this.views.push(view);
    }
    // else, view already exists
}
```

- We'll need a way to remove a view from the tracking array when the user has deleted the underlying Model. Add the following:

```
removeKeyWordView: function (view) {
    var self = this,
        pos = -1,
        i = self.views.length;

    while (i--) {
        if (self.views[i].cid === view.cid) {
            pos = i;
            break;
        }
    }

    if (pos > -1) {
        // remove, if found
        self.views.splice(pos, 1);
    }

}
```

- When the keywords Collection sees a change to one of its members, we'll need a method

that triggers the redrawing of the graphs for each of the views (thus adjusting the bar graphs of the other models as one grows larger). Add the following:

```
// update each view in the list with the new value and bar graph width
updateViews: function () {
    var index = this.views.length,
        view;
    while (index--) {
        view = this.views[index];
        view.updateDisplayValues.call(view);
    }
}
```

Now we need to add the bindEvents function, where we'll set the behavior.

- Add the following:

```
bindEvents: function () {
    var self = this;
    // start of page functionality
    // first, lets see if any keywords actual exist
    this.reloadKeywords(false);

    // keyload the keywords if we have saved a new one (look at add_keyword_container.js for the origin of the event)
    TM.instance.viewManager.views.addContainer.on("saved", function () {
        self.reloadKeywords.call(self, true);
    });
    // if we have no keyword models, show an empty message
    this.keywords.on("empty", function () {
        self.showEmptyMessage();
    });
    // listen for a change event from the collection; update each view... this way, the relative size of the bar graph
    // will update correctly... say, if one keyword is running away with all the hits, the others will adjust their
    // sizes to reflect
    this.keywords.on("change", function () {
        self.updateViews.call(self);
    });

    this.keywords.on("destroy", function (keyword) {

        self.removeKeyWordView(keyword.attachedView);

    });
}
```

During the bindEvents, we listen for notifications from the collection whenever its empty, an item has changed value or an item has been deleted (destroyed) and act accordingly.

In the end, you should have:

```
TM.Views.KeywordContainer = Backbone.View.extend({

    initialize: function () {
        this.keywords = new TM.Collections.Keywords();
        this.keywords.bindEvents();
        this.views = [];
    },

    render: function () {

        $(this.el).html(TM.Templates.keywordContainer({}));
        return this;
    },

    bindEvents: function () {
        var self = this;
        // start of page functionality
        // first, lets see if any keywords actual exist
        this.reloadKeywords(false);

        // keyload the keywords if we have saved a new one (look at add_keyword_container.js for the origin of the event)
```

```
        TM.instance.viewManager.views.addContainer.on("saved", function () {
            self.reloadKeywords.call(self, true);
        });
        // if we have no keyword models, show an empty message
        this.keywords.on("empty", function () {
            self.showEmptyMessage();
        });
        // listen for a change event from the collection; update each view... this way, the relative size of the bar graph
        // will update correctly... say, if one keyword is running away with all the hits, the others will adjust their
        // sizes to reflect
        this.keywords.on("change", function () {
            self.updateViews.call(self);
        });

        this.keywords.on("destroy", function (keyword) {

            self.removeKeyWordView(keyword.attachedView);

        })

    },

    // update each view in the list with the new value and bar graph width
    updateViews: function () {
        var index = this.views.length,
            view;
        while (index--) {
            view = this.views[index];
            view.updateDisplayValues.call(view);
        }
    },

    // triggers the collection's fetch call, then triggers the rendering of views to the screen
    // @param add Determine whether or not to 'add' new elements rather than reset the whole collection
    //
    reloadKeywords: function (add) {
        var self = this;
        this.keywords.fetch({
            add: add,
            success: function (collection, data) {
                self.populateKeywords.call(self, collection, data);
            }
        });
    },

    populateKeywords: function (collection, data) {
        var self = this;

        if(self.views.length === 0) {
            this.$el.html("");
        }
        //underscore.js's each iterator function
        _.each(collection.models, function (model) {
            self.createView.call(self, model);
        });

        //display empty message
        if (collection.models.length === 0){
            this.showEmptyMessage()
        }
    },

    showEmptyMessage: function () {
        this.$el.html(TM.Templates.keywordContainerEmpty());
    },

    createView: function(model) {
        //first, ensure the view hasn't already been created

        if (!model.attachedView) {
            var view = new TM.Views.Keyword({model:model});
            //render it initially
            this.$el.append($(view.render().el));
            //set the element on the new keyword
            view.setElement(this.$el.children().last());
            //and bind!
            view.bindEvents();
            //and store. We'll need to access the view object's reference later for destruction
            this.views.push(view);
        }
        // else, view already exists
    },

    removeKeyWordView: function (view) {
        var self = this,
```

```
            pos = -1,
            i = self.views.length;

        while (i--) {
            if (self.views[i].cid === view.cid) {
                pos = i;
                break;
            }
        }

        if (pos > -1) {
            // remove, if found
            self.views.splice(pos, 1);
        }

    }

});
```

Finally, the Tweet container. This section will generate views of Tweet models that the server sends. Part of the behavior is to repeatedly fade and remove/destroy the oldest Tweet, 'pushing' the others up. It will also ask for more Tweets from the server when 'low' on Models.

- Open web-app/js/src/views/tweet-container.js
- Add:

```
TM.Views.TweetContainer = Backbone.View.extend({

    initialize: function () {
        //max tweets to render at a time, although several (the css overflow) will be hidden
        this.NUM_RENDER = 10;
        //number that once crossed triggers the service to look for more tweets
        this.LOW_THRESHOLD = 10;
        // array to track the views without having to re-query each time
        this.views = [];
        this.lastTweetId = -1;
        // lock to prevent fetching twice
        this.fetching = false;

        //Time for a tweet to display before fading
        this.tweetLiveTime = 1600;
    },


    render: function () {

        $(this.el).html(TM.Templates.tweetContainer({}));
        return this;
    },

    // will attempt to pull tweets from the instance queue and build views from them
    createTweetViews: function () {
        var i,
            max,
            tweets = TM.instance.tweets;

        // clear out empty message
        if (this.views.length === 0 && tweets.length > 0) {
            this.$el.html("");
        }
        //will usually be 1 if things are flowing as planned
        max = this.NUM_RENDER - this.views.length;
        if (max > tweets.length) {
            max = tweets.length;
        }

        for (var i = 0; i < max; i++) {
            this.createTweetView(tweets);
        }

    },

    // creates an individual view from the oldest tweet in the queue.
```

```
    createTweetView: function (tweets) {
        var model = tweets.shift(),
            view = new TM.Views.Tweet({model:model});
        //track the last seen id;
        this.lastTweetId = model.id;
        this.$el.append(view.render().el);
        // setElement
        view.setElement(this.$el.find(".tweet").last());
        this.views.push(view);
        // finally check how many tweets are in the queue. If at threshold, get more
        if (tweets.length === this.LOW_THRESHOLD) {
            this.trigger("start");
        }

    },

    fetchTweets: function () {
        // if the container has our hidden class, prevent tweets from fetching.

        var tweets = TM.instance.tweets,
            self = this;
        self.fetching = true;

        $.ajax({
            url:"/twitterMonitor/tweet/listBatch",
            data: {
                id: self.lastTweetId > 0 ? self.lastTweetId : null
            },
            type:"GET",
            success: function (data) {
                var max = data.length
                for(var i = 0; i < max; i ++) {
                    TM.instance.tweets.push(new TM.Models.Tweet(data[i]));
                }
                // if no views are present, alert the container that it's time to start rendering
                if (self.views.length === 0) {
                    self.trigger("tweetsReceived");
                }
                self.fetching = false;
            },
            failure: function (data) {
                self.fetching = false;
            }
        });

    }

});
```

The above adds in our now familiar initialize() and render() functions, as well as code to control the fetching and rendering of individual Tweet views.

- Now for the fade-out effect. Add:

```
// sets up an interval which will fade and remove the topmost tweet
startTweetFade: function () {
    var self = this;
    self.tweetFadeInterval = setInterval(function () {
        self.fadeOldestTweet.call(self);
    }, self.tweetLiveTime);

},
// clears the interval
stopTweetFade: function () {
    clearInterval(this.tweetFadeInterval);
},

// fades and removes the 'oldest' tweet
fadeOldestTweet: function () {
    var view,
        self = this;

    if (self.views.length > 0) {
        self.views[0].$el.fadeOut("slow", function () {
            view = self.views.shift();
            // the following 2 lines are a technique for deleting a backbone object
            view.unbind();
            view.remove();
```

```
            if (self.views.length < self.NUM_RENDER) {
                self.createTweetViews.call(self);
            }
        });
    }
}
```

This bit of code adds an interval which triggers a function (fadeOldestTweet()).

- Time for bindEvents(). We need to add eventListeners for the various creation and fetching methods defined above. Add:

```
bindEvents: function () {

    var self = this;

    self.on("tweetsReceived", function () {
        self.createTweetViews.call(self);
    });

    self.on("start", function () {
        //only allow fetching if we're not already fetching, and the 'go' toggle has been switched on
        if (!self.fetching && TM.instance.kickingIt) {
            self.fetchTweets.call(self);
        }
    });
};
```

There's a bit more yet to add to this View; we'll revisit it later.

At this point, we have something we can interact with! Start up the grails app and navigate locally to http://localhost:8080/twitterMonitor or http://localhost:8080/twitterMonitor/standAlone to view the application. Start the fetching by clicking the 'kicking it' toggle switch.

If everything goes according to plan, you can begin adding keywords via the text input at the top of the screen. Once you've entered at least one, the Grails server will begin querying Twitter, and any found tweets will be displayed in the Tweet Container.

Note that all of this markup is being generated by the client, and very little actual data is being sent by the server! Fun!

5. Template Helpers

Let's start sprucing things up a little. First, take a look at the text of your tweets as they appear. A bit bland, no? Let's add some color to the text by highlighting the tags and mentions (bits of text that begin with '#' and '@', respectively). We can accomplish this by using a feature in Handlebars called 'Helpers', which are similar to taglibs in Grails. There are several existing Helpers, like iterators and conditionals, but one can easily create their own.

To use a Helper, one creates a special function (which can accept an argument) that should

return a bit of text to render. The function is then registered with Handlebars so it can understand the reference during Template compilation.

To illustrate this, open the file web-app/js/src/helpers/tweet_text_decorator.js. The function 'tweetTextDecorator' accepts a string and performs a series of replaces via Regular Expressions, with the goal of wrapping the found expression with other strings. In our case, we're wrapping substrings with markup tags. The expressions and the wrapping markup are defined in the 'Resources' object at the beginning of this file.

In order for this function to take effect, we'll need to do two things: register the Helper, and use the Helper in our Template.

- Register the Helper by adding the following to tweet_text_decorator.js:

```
//When registering the Helper, the first parameter is the name that Handlebars
//uses to reference your function in the templates
Handlebars.registerHelper('tweetTextDecorator', TM.Helpers.tweetTextDecorator);
```

- We can now use the string 'tweetTextDecorator' as a Helper in our Templates. Open up grails-app/views/standAlone/_handlebars.gsp.
- Change the HBtweet Template from:

```
<script id="HBtweet" type="text/x-handlebars-template">
    <div class="tweet {{class}}">
        <div class="header">
            <img src="{{imageUrl}}" alt="userImage"/>
            <div class="right">{{userName}}</div>
        </div>
        <div class="text clearfix">{{text}}</div>
    </div>
</script>
```

To:

```
<script id="HBtweet" type="text/x-handlebars-template">
    <div class="tweet {{class}}">
        <div class="header">
            <img src="{{imageUrl}}" alt="userImage"/>
            <div class="right">{{userName}}</div>
        </div>
        <div class="text clearfix">{{{tweetTextDecorator text}}}</div>
    </div>
</script>
```

With this change the text of the Tweet, as relayed to us from the server, will be passed through our Helper function before being rendered to the screen. Look carefully at what we've just done, though. See anything different? That's right, the helper block is wrapped with three curly braces instead of two!

Using two curly braces tells Handlebars to simply render the string it receives for that block. By default, it will escape any HTML characters. Normally, this is great, however, we *want* the HTML characters to be rendered as markup. Using three curly braces tells Handlebars to do just that.

Now, reload the page within your browser and start it up. Tweets that appear should have a bit of coloring, which is great. But what about when a user has a url in their tweet?

- Bonus: update the Helper and your CSS to add a link tag around a url in a tweet!

6. Adding a little Responsiveness

Next, the goal is to add a little bit of Responsive Design to this application (note that, ideally, you would plan this from the beginning and incorporate the design during the building of your app). We've already added a bit of css to adjust some of the components based on browser window size.

- Play around with the browser size. Start with a large browser window and drag your browser to be more narrow. As the width of the window decreases, note that the Tweet Container location is placed underneath (the views are now 'stacked'), the text size adjusts, and the Gr8conf logo disappears (sorry).
- You can also view this in your mobile phone's browser, although we apoligize ahead of time for any inconsistancies. This small app was developed in chrome and we had very little QA time!
- These adjustments are done via CSS, using @media queries. To examine this functionality, open up web-app/css/main-responsive.css. The pixel widths in our queries are not indicitve of any particular device, but merely as a tool to demonstrate how to use the queries.

Respnsive Design is more than just adjusting elements in your markup via @media queries, however. Some aspects of it are fairly philisophical, but in our case, let's assume that we consider the display of the actual Tweet text to be secondary to the keyword count display. For our smaller devices, we hide the Tweet container and stop the actual fetching of tweets from the server.

Thus, the Client asks for only for what it needs rather than having the Server sending everything to the Client.

To accomplish this in our small application, we will add some code to the tweetContainer that monitors device width. When a certain size threshold is crossed, the application will hide the tweetContainer and prevent it from fetching tweets.

- Reopen web-app/js/src/views/tweet_container.js
- Add the following to the end of the initialize function:

```
  //Threshold, in px,  above which to display the tweets
```

```
this.displayThreshold = 700;
// class that the view will look for to stop searching; we could also set a flag internally,
// but this does double duty by hiding the view as well
this.hideTweetClass = "verboten";
```

- Add the following two functions:

```
// Checks the innerWidth of the window, and adds a class to this view if the width is under a certain threshold
// will remove the class once the threshold is crossed again
visibilityCheck: function () {
    var width = window.innerWidth,
        hasClass = this.$el.hasClass(this.hideTweetClass);

    if (width <= this.displayThreshold && !hasClass) {

        this.el.classList.add(this.hideTweetClass);
        this.stopTweetFade();

    } else if (width > this.displayThreshold && hasClass){

        this.el.classList.remove(this.hideTweetClass);
        this.startTweetFade();
        // also, alert the container that we can begin receiving tweets again
        if (TM.instance.tweets.length === 0) {
            this.trigger("start");
        }
    }
},

// convenience function to check if the hideTweetClass has been attached; if not we allow fetching
allowFetching: function () {
    var allow = true;
    if (this.$el.hasClass(this.hideTweetClass)) {
        allow = false;
    }
    return allow;
}
```

- Update the 'fetchTweets()' function, wrapping the code already in the function with an if statement that checks 'allowFetching', like so :

```
fetchTweets: function () {
    // if the container has our hidden class, prevent tweets from fetching.
    if (this.allowFetching()) {

        ... Existing code!

    }
}
```

- Update the render function to execute visibilityCheck, in case the browser starts at a small size (e.g. your phone):

```
 render: function () {

    $(this.el).html(TM.Templates.tweetContainer({}));

    this.visibilityCheck();

    // Backbone convention is to return this from render()
    return this;
}
```

- Finally, update bindEvents to listen for window.resize():

```
bindEvents: function () {

    var self = this;

    self.on("tweetsReceived", function () {
        self.createTweetViews.call(self);
    });
```

```
    self.on("start", function () {
        //only allow fetching if we're not already fetching, and the 'go' switch has been switched on
        if (!self.fetching && TM.instance.kickingIt) {
            self.fetchTweets.call(self);
        }
    });

    if (self.allowFetching()) {
        self.startTweetFade();
    }

    // we want to only show the tweets if the browser window is above a certain threshold
    window.addEventListener("resize", function () {
        self.visibilityCheck();
    });
}
```

- In the end, your tweet_container.js should look something like:

```
TM.Views.TweetContainer = Backbone.View.extend({

    initialize: function () {
        //max tweets to render at a time, although several (the css overflow) will be hidden
        this.NUM_RENDER = 10;
        //number that once crossed triggers the service to look for more tweets
        this.LOW_THRESHOLD = 10;
        // array to track the views without having to re-query each time
        this.views = [];
        this.lastTweetId = -1;
        // lock to prevent fetching twice
        this.fetching = false;

        //Time for a tweet to display before fading
        this.tweetLiveTime = 1600;
        //Threshold, in px,  above which to display the tweets
        this.displayThreshold = 700;
        // class that the view will look for to stop searching; we could also set a flag internally,
        // but this does double duty by hiding the view as well
        this.hideTweetClass = "verboten";
    },

    render: function () {

        $(this.el).html(TM.Templates.tweetContainer({}));

        this.visibilityCheck();

        // Backbone convention is to return this from render()
        return this;
    },

    bindEvents: function () {

        var self = this;

        self.on("tweetsReceived", function () {
            self.createTweetViews.call(self);
        });

        self.on("start", function () {
            //only allow fetching if we're not already fetching, and the 'go' switch has been switched on
            if (!self.fetching && TM.instance.kickingIt) {
                self.fetchTweets.call(self);
            }

        });

        if (self.allowFetching()) {
            self.startTweetFade();
        }

        // we want to only show the tweets if the browser window is above a certain threshold
        window.addEventListener("resize", function () {
            self.visibilityCheck();
        });
    },

    // Checks the innerWidth of the window, and adds a class to this view if the width is under a certain threshold
    // will remove the class once the threshold is crossed again
    visibilityCheck: function () {
        var width = window.innerWidth,
```

```
            hasClass = this.$el.hasClass(this.hideTweetClass);

        if (width <= this.displayThreshold && !hasClass) {

            this.el.classList.add(this.hideTweetClass);
            this.stopTweetFade();

        } else if (width > this.displayThreshold && hasClass){

            this.el.classList.remove(this.hideTweetClass);
            this.startTweetFade();
            // also, alert the container that we can begin receiving tweets again
            if (TM.instance.tweets.length === 0) {
                this.trigger("start");
            }
        }
    },

    allowFetching: function () {
        var allow = true;
        if (this.$el.hasClass(this.hideTweetClass)) {
            allow = false;
        }
        return allow;
    },

    // sets up an interval which will fade and remove the topmost tweet
    startTweetFade: function () {
        var self = this;
        self.tweetFadeInterval = setInterval(function () {
            self.fadeOldestTweet.call(self);
        }, self.tweetLiveTime);

    },
    // clears the interval
    stopTweetFade: function () {
        clearInterval(this.tweetFadeInterval);
    },

    // fades and removes the 'oldest' tweet
    fadeOldestTweet: function () {
        var view,
            self = this;

        if (self.views.length > 0) {
            self.views[0].$el.fadeOut("slow", function () {
                view = self.views.shift();
                // the following 2 lines are a technique for deleting a backbone object
                view.unbind();
                view.remove();

                if (self.views.length < self.NUM_RENDER) {
                    self.createTweetViews.call(self);
                }
            });
        }
    },

    // will attempt to pull tweets from the instance queue and build views from them
    createTweetViews: function () {
        var i,
            max,
            tweets = TM.instance.tweets;

        // clear out empty message
        if (this.views.length === 0 && tweets.length > 0) {
            this.$el.html("");
        }
        //will usually be 1 if things are flowing as planned
        max = this.NUM_RENDER - this.views.length;
        if (max > tweets.length) {
            max = tweets.length;
        }

        for (var i = 0; i < max; i++) {
            this.createTweetView(tweets);
        }

    },

    // creates an individual view from the oldest tweet in the queue.
    createTweetView: function (tweets) {
        var model = tweets.shift(),
            view = new TM.Views.Tweet({model:model});
        //track the last seen id;
```

```
            this.lastTweetId = model.id;
            this.$el.append(view.render().el);
            view.setElement(this.$el.find(".tweet").last());
            this.views.push(view);
            // finally check how many tweets are in the queue. If at threshold, get more
            if (tweets.length === this.LOW_THRESHOLD) {
                this.trigger("start");
            }

        },

        fetchTweets: function () {
            // if the container has our hidden class, prevent tweets from fetching.
            if (this.allowFetching()) {
                var tweets = TM.instance.tweets,
                    self = this;
                self.fetching = true;

                $.ajax({
                    url:"/twitterMonitor/tweet/listBatch",
                    data: {
                        id: self.lastTweetId > 0 ? self.lastTweetId : null
                    },
                    type:"GET",
                    success: function (data) {
                        var max = data.length
                        for(var i = 0; i < max; i ++) {
                            TM.instance.tweets.push(new TM.Models.Tweet(data[i]));
                        }
                        // if no views are present, alert the container that it's time to start rendering
                        if (self.views.length === 0) {
                            self.trigger("tweetsReceived");
                        }
                        self.fetching = false;
                    },
                    failure: function (data) {
                        self.fetching = false;
                    }
                });
            }
        }
    });
```

Reload the application in your browser, monitor the traffic from your browser to the server. As your resize the window to be narrow, the tweetContainer should dissappear, and no more traffic should be made to the /listBatch endpoint on the server

7. Final Task(s)

Congratulations! We hope you've learned a bit about working with Javascript-based UIs, and possibly enjoyed yourself in the process. If you're up to it, we have a few more tasks for you:

- Update the TweetTextHelper to highlight urls (or other symbols, too)
- Update the TweetContainer and Tweet Views to use a Collection

Good luck!