Backbone - Day 2

Macy's Learning Spike

Useful links

- Repository for all labs code + solutions: https://github.com/alcfeoh/di-backbone-js

- Link to these slides: https://goo.gl/vkEiye

Outline for today

Backbone Fetch and Events

Backbone Routing

Marionette Architecture

Marionette Views

Testing Backbone

Outline for today

Backbone Fetch and Events

Backbone Routing

Marionette Architecture

Marionette Views

Testing Backbone

Backbone Fetch and Events

Fetch and REST

Backbone is pre-configured to sync with a RESTful API

Both models and collections can use that feature to interact with the server

A collection expects an array from the server while a model expects an object

```
var Books =
Backbone.Collection.extend({
    url: '/books'
});
```

```
GET  /books/ .... collection.fetch();
POST /books/ .... collection.create();
GET  /books/1 ... model.fetch();
PUT  /books/1 ... model.save();
DEL  /books/1 ... model.destroy();
```

Backbone.sync

- **Backbone.sync** is the function that Backbone calls every time it attempts to read or save a model to the server

- Whenever a sync starts, a "**request**" event is emitted. If the request is successful you'll get a "**sync**" event, and an "**error**" event if not

  - **"request"** (model_or_collection, xhr, options) — when a model or collection has started a request to the server.
  - **"sync"** (model_or_collection, response, options) — when a model or collection has been successfully synced with the server.
  - **"error"** (model_or_collection, response, options) — when a model's or collection's request to the server has failed.

Lab 7 - Fetching data from a server

- In this lab, we're going to update our store app to get its data from a server.
- First, open a terminal in the **server** directory of your project. Run **`npm install`** then **`node server.js`**. A REST API now runs on port **8000**
- We will use that server to serve our Backbone app as well. Copy-paste both **app.js** and **index.html** into **server/ui**
- **Your mission:** Start from the directory **`7-fetch-plates`**. Update our collection so that it gets linked to the url: **/data**
- Then when the main view gets initialized, ask the collection to fetch its data.
- You can now remove the hardcoded list of plates from your Javascript code.

Events

Events is a module that can be mixed into any object, giving the object the ability to bind and trigger custom named events.

```
var object = {};

_.extend(object, Backbone.Events);

object.on("alert", function(msg) {
    alert("Triggered " + msg);
});

object.trigger("alert", "an event");
```

Events

Events can be registered on a view as illustrated here

All DOM events are supported. As you can see here, events can be bound to any HTML element of the view. Here we bind to specific CSS classes.

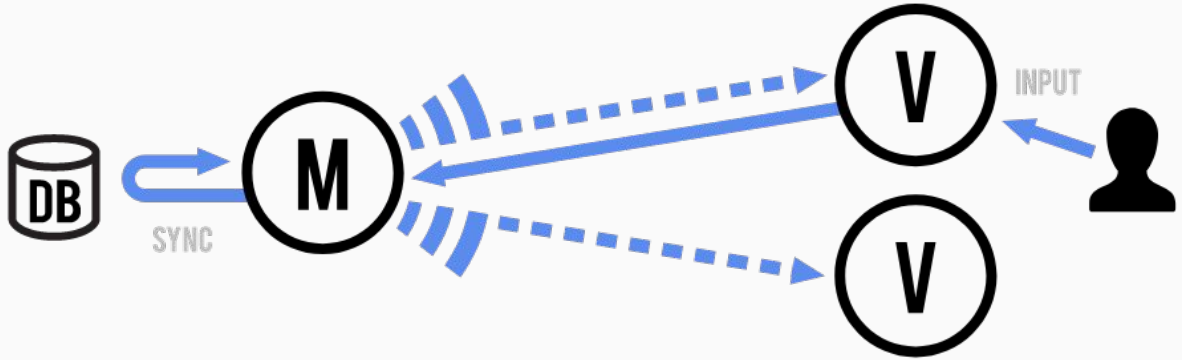
```
var TodoView = Backbone.View.extend({
    //...

    events: {
        // When click on .toggle element...
        "click .toggle"   : "toggleDone",
        "dblclick .view"  : "edit",
        "blur .edit"      : "close"
    },
    toggleDone: function() {
        this.model.toggle();
    },
    edit: function() {
        this.$el.addClass("editing");
        this.input.focus();
    }
```

Events

A view can also listen to model updates so that it can refresh its template accordingly.

This is achieved with the **`listenTo`** function, passing the model as a first parameter, then the event name, then a callback function.

```
initialize: function() {
    this.listenTo(this.model, 'change',
                  this.render);
    this.listenTo(this.model, 'destroy',
                  this.remove);
},
```



Lab 8 - Listening to user events

- In this lab, we're going to update our store app so that items can be added to the cart.
- **Your mission:** Start from the directory **`8-add-to-cart`**. Create a new model **`CartItem`** that uses the url: **`/cart`**
- Register an event so that when the "Add to cart" button of a **`LicensePlateView`** gets clicked, we create a new **`CartItem`** and persist it to the server.
- The **`CartItem`** data model will be the same as the one from **`LicensePlate`**
- **Note:** You can see the contents of the cart with **`HTTP GET /cartContents`**

What we just learnt

Backbone can register event listeners on the DOM or on any kind of object to perform model or view actions

Backbone is pre-configured to sync with a RESTful API

A view can also listen to model updates so that it can refresh its template accordingly

Outline for today

Backbone Events

Backbone Routing

Marionette Architecture

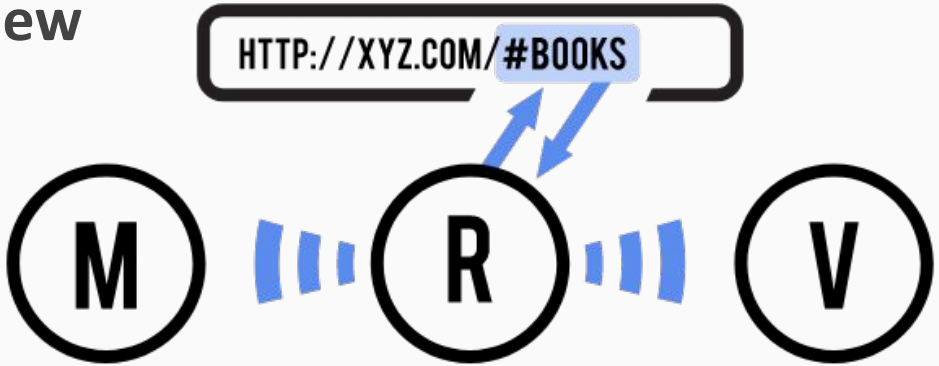
Marionette Views

Testing Backbone

Backbone Routing

How to use URLs to keep track of the state of an app?

- In a browser, URLs can be bookmarked or shared
- This means that front-end code should be able to restore a specific state based on the browser URL
- The **Backbone Router** allows this by pairing routes to actions
- For instance, **/store** would load a **StoreView** on the screen, and **/cart** would load a **CartView**



Router

Routes can be defined as triggers that would call a function.

For instance, navigating to **`/help`** would call the **`help`** function

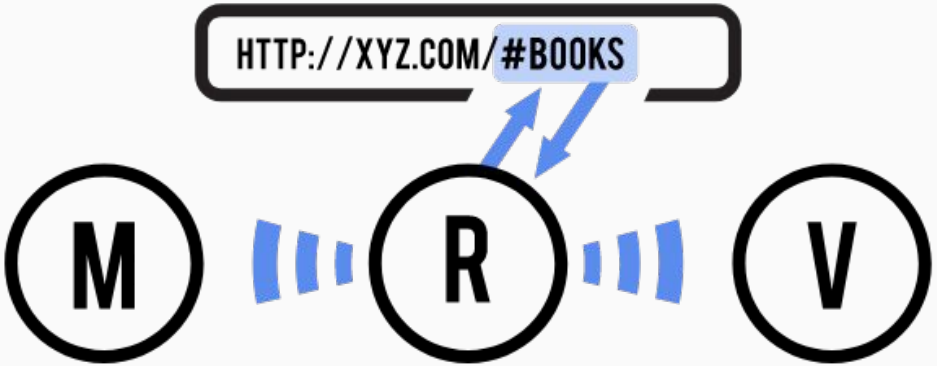
Parameters can also be added to the route path and used in the triggered function

```
var Router = Backbone.Router.extend({

    routes: {
      "help": "help",
      "search/:query": "search",
      "search/:query/:page": "search"
    },

    help: function () {
        //...
    },

    search: function (query, page) {
        //...
    }
```

Browser history

- During page load, after your application has finished creating all of its routers, be sure to call **`Backbone.history.start()`** to route the initial URL.

- That way, users will be able to use the back / forward buttons of the browser to navigate back and forth in history



Events and Navigation

Event listeners can be registered anywhere to listen to route changes

The router can also be used to navigate programmatically using the **navigate** method

```
router.on("route:help", function(page) {
  //...
});


// Updates browser URL
// and triggers the route function
router.navigate("help/troubleshooting",
                          {trigger: true});

//Or ...
// Updates browser URL,
// triggers route function AND
// replaces current route in browser history
router.navigate("help/troubleshooting",
                 {trigger: true, replace: true});
```

Lab 9 - Routing

- In this lab, we're going to update our store app so that it supports two routes.
- **Your mission:** Start from the directory **`9-route-to-cart`**. Create a router that has two route definitions: **`/store`** and **`/cart`**
- **`/store`** will display the current view with all of the store items
- **`/cart`** should only render the items in the cart, using the same HTML **`#container`**
- Update the navigation links at the top of the webpage to easily navigate between these two views

What we just learnt

Backbone has a router that can be used to trigger specific views and models based on the browser URL

Backbone Router has its own set of events

It allows for back and forth navigation through the browser history

Navigation can be done programmatically with the **`navigate`** method

Outline for today

Backbone Events

Backbone Routing

Marionette Architecture

Marionette Views

Testing Backbone

Marionette Architecture

How to build Backbone applications?

- Backbone is unopiniated, which is good but also leaves the door open to too many options
- As we saw during our previous labs, we often get to a point where we don't know how to architect things
- For instance, what to do in a router function? How to do it? Backbone does not attempt to answer those questions
- The same goes for views: The render function does not do anything, we have to decide how to render the view.

Enter Marionette

- That's where Marionette comes into play. Unlike Backbone, it is opinionated and decides how things should be done.

- Marionette uses Backbone

- It can be seen as an additional layer

on top of Backbone, which gets manipulated like a puppet, hence the name:

Problems that Marionette tries to solve

- How to render Views?
- How to manage relationships between objects?
- How to make Views communicate?
- How to structure our application?
- How to prevent memory leaks?

Structure: Application

Provides a single entry point to render our application

```
var App = Marionette.Application.extend({
    region: '#root-element',

    onStart: function() {
        this.showView(new RootView());
    }
});

var myApp = new App();
myApp.start();
```

Rendering: View

Views use underscore by default

No need to implement the **`render()`** function anymore!

```
var MyView = Marionette.View.extend({
    tagName: 'h1',
    template: '#template'
});

var myView = new MyView();
myView.render();
```

Communication: Radio

Radio is an event manager where we can send and listen to events

This allows views to communicate

```
var inboxChan = Backbone.Radio.channel('inbox');

var ContactView = Marionette.View.extend({

   template: '#contact-template',

   initialize: function() {
     this.listenTo(inboxChan, 'show:email',
                                this.showContact);
     this.listenTo(inboxChan, 'show:inbox',
                                  this.showAd);
    },
    showContact: function(emailObject) {
            //…
    },
    showAd: function() {
            //…
    }
```

Lab 10 - Hello Marionette

- In this lab, we're going to create a simple **`HelloMarionette`** app
- **Your mission:** Start from the directory **`9-route-to-cart`**. Create a router that has two route definitions: **`/store`** and **`/cart`**
- **`/store`** will display the current view with all of the store items
- **`/cart`** should only render the items in the cart, using the same HTML **`#container`**
- Update the navigation links at the top of the webpage to easily navigate between these two views

What we just learnt

Marionette provides additional structure to Backbone so that developers have less decisions to make

Radio is a way to achieve communication between views

Application provides a single entry point for our app

Views automatically render **underscore** templates

Outline for today

Backbone Events

Backbone Routing

Marionette Architecture

Marionette Views

Testing Backbone

Marionette Views

View

Views use underscore by default

No need to implement the **`render()`** function anymore!

```
var MyView = Marionette.View.extend({
    template: '#template'
});

var myView = new MyView();
myView.render();
```

Collection View

Automatic rendering of a collection of models applied to child views.

No need to provide any **`render`** or **`initialize`** method!

```
var TodoListView =
Marionette.CollectionView.extend({
    childView: TodoView,
    collection: todoCollection,
});
```

Regions

Regions are areas that you can define to render specific views.

Makes it easy to architect your application and swap views in some areas when needed.

```
var RootView = Marionette.View.extend({

    regions: {
        header: '#navbar',
        footer: 'footer'
    },

    initialize: function() {
        this.getRegion('header')
                        .show(new HeaderView());
        this.getRegion('footer')
                        .show(new FooterView());
    }
});
```

View Lifecycle

All of these events are triggered during the view lifecycle.

You can implement a handler for each of them, for instance: **`onDetach()`** would be called when the **`detach`** event happens

```
Before:render  // Before rendering el
Render    // el is ready, not in the DOM yet
Before:attach  // Before first DOM rendering
Attach    // el is in the DOM
Dom:refresh // every time render() is called
Before:destroy   // Before destroying
Before:detach // Before removing from DOM
Dom:remove // every time render() is called
Detach  // el removed from DOM
Destroy // View is completely gone
```

Lab 11 - Marionette Views

- In this lab, we're going to use Marionette for our License Plate Store.
- **Your mission:** Start from the directory **`10-full-marionette`**. Create a **`Marionette.Application`** to start the app.
- Create a **`Marionette.View`** to render a license plate
- Create a **`Marionette.CollectionView`** to render our collection of license plate
- We won't use the router here to make it easier on you :-)

What we just learnt

Marionette Views bring new features to Backbone that remove a lot of boilerplate code

Templates are automatically rendered

`CollectionViews` automatically iterate through and render their child views

Regions are a good way to handle different areas of our app and decide which view should render where.

Outline for today

Backbone Events

Backbone Routing

Marionette Architecture

Marionette Views

Testing Backbone

Testing Backbone

Jasmine

Jasmine is Behavior Driven Development framework for testing Javascript applications

Official website: https://jasmine.github.io/

```
describe("A suite is just a function",
  function() {
    var a;

    it("and so is a spec", function() {
        a = true;
        expect(a).toBe(true);
    });
});
```

Jasmine BeforeEach

BeforeEach initializes the context of each test

Then each test is an **it()** function that runs an action and expects a result with the **expect** function and assertions

```
describe("Player", function() {
 var player;
 var song;

 beforeEach(function() {
   player = new Player();
   song = new Song();
 });

 it("should be able to play a Song", function() {
   player.play(song);
   expect(player.currentlyPlayingSong)
                                        .toEqual(song)
 });
```

Jasmine Spies

Spies are an easy way to mock specific pieces of code for testing purposes.

For instance, this example replaces the **`fetch()`** function with a fake one that sets testing data to the model so we can test without making HTTP requests

```
spyOn(todoCollection, "fetch")
        .and.callFake(function() {
            //Set fake data for testing
            todoCollection.model = [...];
        });
```

Lab 12 - Testing Backbone

- In this lab, we're going to write a couple of tests.
- **Your mission:** Start from the directory **`jasmine-2.8.0`**. Use **`SpecRunner.html`** to see the sample tests in action.
- Then create your own test spec that will mock the **`fetch`** method of our collection to return two fake license plate instead (you can get that data from one of our early labs)
- Then make sure that the **`CollectionView`** will have two **`LicensePlate`** models to render, and that the data from those models is what we expect (same title, same picture, etc.)

What we just learnt

Jasmine is a simple test framework that makes testing Jasmine applications easy

Jasmine runs tests specifications with expectations and outputs a report.

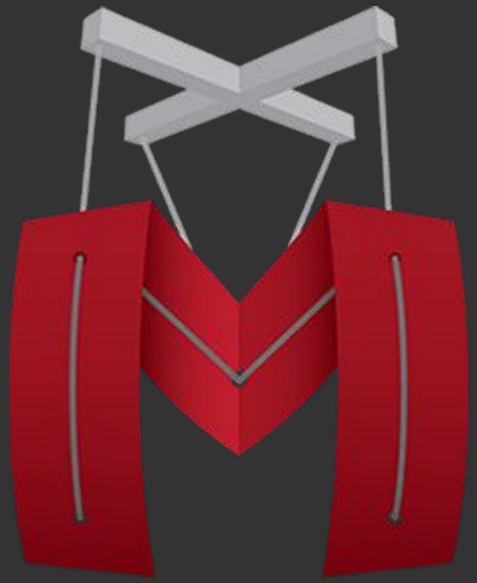
Spies are a way to mock pieces of code in order to return test-doubles for testing purposes

Thanks for your attention

I need your feedback before you leave:

https://www.surveymonkey.com/r/LDXJK32

BACKBONE.JS



Marionette.js