Introduction to Backbone.js

Table of Contents

Course Objectives

1. Understand the primary purpose of Backbone and the web development problems that it solves
2. Explore the basics of the five core objects of Backbone: Models, Collections, Views, Routers, and Events
3. Interact with each object type observing how they work and understanding why they work the way they do
4. Know when and how to utilize each object within a web application

Overview

Backbone is a JavaScript library the allows developers to create rich client-side web applications without locking them in to an opinionated framework that limits their choices concerning templates, data binding, web components, and other aspects of an application's structure. Because Backbone is not strongly opinionated, and it provides only a few of the capabilities needed for a single page application, it's called a library not a framework. Frameworks, such as Angular or Ember, provide more built-in features than Backbone.js, but do so at the cost of freedom and flexibility thereby making them more opinionated.

Backbone provides five basic building blocks to your application: models, collections, views, routing and events. In the course, we will learn and explore each of these component independently and in relation to each other.

To use Backbone, two JavaScript files need to be included in your web page. They can be referenced from a CDN as follows:

```
<!-- Content Delivery Network (CDN) -->
```

```
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/underscore.js/1.7.0/underscore.js">
</script>
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/backbone.js/1.1.2/backbone.js">
</script>
```

Or, from a local server folder in the web application.

```
<!-- Web Application Folders -->
<script src="libs/underscore.js"></script>
<script src="libs/backbone.js"></script>
```

Underscore is the only hard dependency required by Backbone. Underscore is a very useful library for working with objects, arrays, collections and such. Even if the web application does not use Backbone, using the Underscore library often proves to be very helpful.

Underscore.js is written and maintained by the same author of Backbone.js, Jeremy Ashkenas. He distributes them as separate libraries so developers can use the functionality of Underscore.js without Backbone.js. Sometimes this is misrepresented as Backbone.js having external dependencies when really that is not a fair description.

To learn more about Underscore, browse to the following web site: http://underscorejs.org/.

To learn more about Backbone, browse to the following web site: http://backbonejs.org/.

To learn more about Jeremy, visit his Github page: https://github.com/jashkenas.

Additionally, there are two JavaScript libraries that are commonly used with Backbone to facilitate DOM selection and manipulation, AJAX calls and better template services. For DOM selection and manipulation as well as AJAX calls, jQuery is generally included as shown:

```
<!-- Content Delivery Network (CDN) -->
<script
 src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.3/jquery.js">
</script>
```

Or, from a local server folder in the web application.

```
<!-- Web Application Folders -->
<script src="libs/jquery.js"></script>
```

While Underscore provides a basic template engine, many developers prefer to use the Handlebars template engine. The template engine used in this course will be Handlebars. To include Handlebars, add a script reference to your HTML file as shown:

```
<!-- Content Delivery Network (CDN) -->
<script
  src="https://cdnjs.cloudflare.com/ajax/libs/handlebars.js/3.0.0/handlebars.js">
</script>
```

Or, from a local server folder in the web application.

```
<!-- Web Application Folders -->
<script src="libs/handlebars.js"></script>
```

Events

Objectives

1. Understand the purpose of Backbone Events
2. Learn how to create objects that inherit from Backbone.Events
3. Explore how to trigger and handle custom events
4. Explore how to listen for events on other objects

Introduction

Events are a very popular feature of Backbone. The Events object enables objects, such as objects based on Backbone.Model, to respond to events triggered internally by Backbone.Events or to custom events triggered by application logic.

Add Events to Any Object

In addition to using events with Backbone.Model objects, Backbone.Events can be added to any object. To add Backbone.Events to an object, the **_.extend** function provided by the Underscore library is used.

```
var bankAccount = {
    balance: 0,
    deposit: function(amt) {
        this.balance += amt;
    },
    withdraw: function(amt) {
        this.balance -= amt;
    }
};

_.extend(bankAccount, Backbone.Events);
```

After calling the **_.extend** function, the **bankAccount** object now has full support for Backbone events.

Creating Custom Events

To create a custom event, simply use the **trigger** method to trigger the desired event by name.

```
function withdraw(amt) {
    this.balance -= amt;

    // not enough money
    if (amt < 0) {
        this.trigger("overdrawn", {
            balance: this.balance
        });
    }
}

bankAccount.withdraw = withdraw;
```

Observe how the implementation of the **withdraw** function, calls the **trigger** function provided by the Backbone.Events object prototype to trigger a custom event on the object. Once the event is trigger it then has to be handled.

Handling Custom Events

To handle an event using Backbone.Events, the **on** method is used. Simply, pass in the name of the event as well as a handler function, when the event is triggered using the **trigger** function, the event handler will be invoked with an argument from the event being passed in.

```
bankAccount.on("overdrawn", function(e) {
    console.dir("Account was overdrawn by " + e.balance);
});

bankAccount.deposit(100);
bankAccount.withdraw(200);
```

Here is the output of the above code.

```
Account was overdrawn by -100
```

An interesting side note is that the **on** (and its corresponding **off** function described in the next section) is the common name for functions registering event handlers. Not only does Backbone use this function name but so do other libraries like jQuery.

Removing Event Handlers

To remove event handlers, the **off** function must be called, with the name of the event and a reference to the function being passed in as arguments.

```
function accountOverdrawn(e) {
    console.dir("Account was overdrawn by " + e.balance);
}

bankAccount.on("overdrawn", accountOverdrawn);

bankAccount.deposit(100);
bankAccount.withdraw(200);

bankAccount.off("overdrawn", accountOverdrawn);
```

Observe how in the earlier **on** example in the previous section, the event handler function was passed inline into the **on** function. This approach is sufficient, if we do not intend to remove the event handler. If we intend to remove the handler, then we need to define the handler as a named function, and the pass it as an argument to both the **on** and **off** function.

Additionally, all event handlers for a particular event can be removed by simply specifying the name of the event and not passing a specific event handler function.

```
bankAccount.off("overdrawn");
```

Occasionally, there will be a special situation where you want to attach an event handler to execute one time and then have the handler removed after the first execution of it. This scenario can be accomplished with the **once** function.

```
function accountOverdrawn(e) {
    console.dir("Account was overdrawn by " + e.balance);
}

bankAccount.once("overdrawn", accountOverdrawn);

bankAccount.deposit(100);
bankAccount.withdraw(200);
```

By using the **once** function, the **accountOverdrawn** event handler will only be executed when the first **overdrawn** event is triggered.

Listening to Events on Other Objects

One additional and useful capability of the Backbone.Events object is that one object can listen to the events of other objects. The **listenTo** function is used to listen to the events of another object. The function accepts three parameters: the object to listen to, the event to listen to and the event handler.

```
var dashboard = {};

_.extend(dashboard, Backbone.Events);

dashboard.listenTo(bankAccount, "overdrawn", function(e) {
    console.dir("Dashboard: Account was overdrawn by " + e.balance);
});

bankAccount.deposit(100);
bankAccount.withdraw(200);
```

Observe how the **dashboard** object listens to the **bankAccount** object by registering an event handler for the **overdrawn** event. When the **bankAccount** object triggers the **overdrawn** event, the **dashboard** object will be through the execution of the event handler function that it registered.

Removing Event Listeners for Other Objects

There are three ways in which an object can stop listening to an event. All three ways use the **stopListening** function. Observe the examples of using the **stopListening** function below.

```
dashboard.stopListening(bankAccount);

dashboard.stopListening(bankAccount, "overdrawn");

dashboard.stopListening(bankAccount, "overdrawn", dashboardAccountOverdrawn);
```

The first usage of **stopListening** removes all event handlers for all events from the object. The second usage of **stopListening** removes all event handlers for a specific event from the object. The third usage of **stopListening** removes a specific event handler for a specific event from the object.

Exercise

Step 1. With a text editor, open the file **index.js** file from folder **exercises/events/js**.

Step 2. Using an object literal, create a new object named car. The object should have a property named **speed** and a property named **drive**. Initialize speed to **0** and **drive** to be a function that accepts one parameter named **newSpeed**. Assign **newSpeed** to the object's **speed** property in the function implementation.

Step 3. Using the Underscore **extend** function, copy the properties of the **Backbone.Events** object to the **car** object.

Step 4. When the **newSpeed** is above 80, trigger a custom event named **toofast** passing in the speed and how high it is over 80. When the **newSpeed** is below 50, trigger a custom event named **tooslow** passing in the speed and how much it is lower than 50. Both custom events should output the actual speed to the console.

Step 5. Open the file **exercises/events/index.html** from the file system with the web browser of your choice. Open the JavaScript Console (it's part of the Developer Tools for the browser).

Step 6. Call the **drive** function on the **car** object, passing in a value of 65. Reload the web page. Are any events triggered? Is there any output to the console?

Step 7. Call the **drive** function on the **car** object, passing in a value of 85. Reload the web page. Are any events triggered? Is there any output to the console?

Step 8. Call the **drive** function on the **car** object, passing in a value of 45. Reload the web page. Are any events triggered? Is there any output to the console?

Congrats you have completed the exercise!

Review Questions

1. What is the purpose of the Backbone.Events object?
2. Does the Backbone.Model object inherit from the Backbone.Events object?
3. Can Backbone.Events be used with objects other than Backbone objects?
4. Can custom events be configured with Backbone.Events?

Backbone Models

Objectives

1. Understand the purpose of Backbone Models
2. Learn how to create a model that inherits from Backbone.Model
3. Explore how to work with attributes, functions and events with models.

Introduction

All software applications contain data: data for managing the application itself and data for the user's business or other processes. There are many kinds of data depending upon which part of the application is being considered. There is data which is permanently stored in files or databases. Also, there is data that is stored in various caching mechanisms such as caching services or cache objects. When people think of data they usually think about data such as financial transactions, inventory data, personnel data, and such. In addition to this kind of data, there is also data that is used to manage the application itself such as user logins and passwords, application settings, data connection strings, and such.

User interface libraries like Backbone provide a mechanisms for storing, manipulating and synchronizing data. In Backbone, the mechanism by which data is stored is through the **Model** object. The **Model** object is extended for each kind of data that needs to be stored. Various functions and attributes can be added to these models to store and manipulate the data appropriate to the kind of data being stored. Additionally, Backbone Models support **Events** allowing the changing of data to be observed, and responded to, by other portions of the applications that are interested in those changes.

To create a new model with Backbone, the core Backbone.Model object needs to be extended. Extending the model creates a new object that prototypally inherits from the base Backbone.Model object. To create a new model, review the code below

Create a New Model

```
var Person = Backbone.Model.extend();

var myPerson = new Person();
```

Once the Person model object is created, then a new instance of the Person can be created using the **new** operator.

Model Initialization

Backbone models provide two properties that can be used to initialize the module: initialize and constructor. The initialize function allows for default model attribute values to be set as well as perform other initialization code.

```
var Person = Backbone.Model.extend({
  initialize: function() {
    this.set("firstName", "");
    this.set("lastName", "");
  }
});

var myPerson = new Person();
```

The second initialization property is the constructor property, and it overrides the default constructor function for the object. The constructor function serves the same general purpose as the initialize function, except the base constructor must be called, if you want it to be executed.

```
var Person = Backbone.Model.extend({
  constructor: function() {
    this.set("firstName", "");
    this.set("lastName", "";
    Backbone.Model.apply(this, arguments);
  }
});

var myPerson = new Person();
```

The call to **Backbone.Model.apply**, calls the base Backbone.Model constructor passing in the arguments that were passed to the new constructor function.

A final, and frequently used option, for setting default attribute values is as follows.

```
var Person = Backbone.Model.extend({
  defaults: {
    firstName: "",
    lastName: ""
  }
});

var myPerson = new Person();
```

The **defaults** option allows an object to specified that will specify the default value for each attributed.

Model Attributes

A common operation performed on models is the getting and setting of attributes. Attributes represent the data of the model, and are accessed via the **get** and **set** functions provided by the Backbone.Model object prototype.

```
var Person = Backbone.Model.extend({
  constructor: function() {
    this.set("firstName", "");
    this.set("lastName", "");
    Backbone.Model.apply(this, arguments);
  }
});

var myPerson = new Person();

myPerson.set("firstName", "Bob");
myPerson.set({ lastName: "Smith" });

console.log(myPerson.get("firstName"));
console.log(myPerson.get("lastName"));
```

Observe the two ways in which the **set** method may be called. In the first instance, the **set** method is called with two arguments: name of the attribute and the new value of the attribute. The **set** method will assign the new value to the named attributed for the model. The second **set** method call accepts an object as a parameter. The object is simply an object map of the model attributes and values that need to be set. Calling set once with an object map is more efficient than calling set multiple times, once for each attribute.

Also, model attributes can set when the model is instantiated. Observe the code below.

```
var myPerson = new Person({
    firstName: "Bob",
    lastName: "Smith"
});
```

The attributes **firstName** and **lastName** will be set to the values passed in.

Model Ids

There is a special attribute that can be set on the model named **id**. The following code demonstrates how to set it.

```
var myPerson = new Person({
    id: 1,
    firstName: "Bob",
    lastName: "Smith"
});
```

Unlike other attributes, the special **id** attribute, can be retrieved like this.

```
var personId = myPerson.id
```

The **id** serves as a unique identifier for the model. It's very useful for retrieving models from collections. In some cases, the name of the **id** attribute may need to be changed. To use a different property for the **id** value of the model, the following option can be set.

```
var myPerson = new Person({
    idAttribute: "_id"
});
```

The property name **_id** is a common **id** property name used by MongoDB. To simplify development, developers typically configure Backbone.js models to use the **_id** name for the **attribute** to make working with both Backbone.js and MongoDB easier.

Typically, the **id** value is determined by the permanent storage for the model data. Permanent storage, such as a relational database, usually assigns a unique identifier to the model. Therefore, the **id** is usually not directly set by the JavaScript code.

Model Functions

In addition to attributes, function properties can be added to models too. Functions provide logic for calculating values based upon property attributes, attribute validation, etc... The logic implemented by a function can either be user interface logic or business logic. Implementing a function is very simple as demonstrated below using the function **getFullName**.

```
var Person = Backbone.Model.extend({

  constructor: function() {
    this.set("firstName", "");
    this.set("lastName", "");
    Backbone.Model.apply(this, arguments);
  },

  getFullName: function() {
    return this.get("firstName") + " " + this.get("lastName");
  }

});

var myPerson = new Person();

myPerson.set("firstName", "Bob");
myPerson.set({ lastName: "Smith" });

console.log(myPerson.getFullName());
```

Once the model is created, and it's attributes have meaningful values, the function is called as a property of the model itself and its operations are performed.

Model Events

Backbone models can be configured to respond to events. Events can include many of the builtin events such as when an attribute is changed, or they can be custom events that are triggered by custom user interface logic. Custom events are covered later in the course.

To configure an event handler, the **on** method is used. The **on** method is provided by the **Backbone.Events** object. The **Backbone.Model** object extends the **Backbone.Events** object.

```
var Person = Backbone.Model.extend({

  constructor: function() {
    this.set("firstName", "");
    this.set("lastName", "");
    Backbone.Model.apply(this, arguments);
  },
  getFullName: function() {
    return this.get("firstName") + " " + this.get("lastName");
  }

});

var myPerson = new Person();

myPerson.on("change:firstName", function(model, firstName) {
  console.log("first name changed");
  console.log(model);
  console.log(firstName);
});

myPerson.set("firstName", "Bob");
```

The **on** method above sets an event handler function for when the **firstName** attribute is changed. The event handler is passed the specific model object as well as the property value that was changed.

Model Validation

Backbone.js provides the ability to valid model data through the use of the **validate** property. Consider the following Person model:

```
  var Person = Backbone.Model.extend({

    defaults: {
      firstName: "",
      lastName: "",
      age: 0
    },

    validate: function(attrs, options) {

       if (attrs.age < 13) {
         return "Sorry, you are too young to sign up with this web site.";
       }
    }

  });
```

The **validate** function will be executed when the model's **save** function is executed (requires the sync option to be configured which is beyond the scope of this course). To manually invoke the **validate** function, call the **set** function with the following option.

```
  var p = new Person();

  p.set("age", 12, { validate: true });
```

Passing the **validation** option set to **true** will cause the **validate** function to execute when the value is set. When the **validate** function executes, it should only return a value if the model is invalid. Otherwise, the function should return nothing. The value returned from the **validate** function will be accessible on the **validationError** property.

```
console.log(p.validationError);
```

This value can be a string, as shown in the example code, but its not limited to being a string. Any kind of value or object can be returned. A common scenario is to return an array of validation errors to be displayed in the web page.

Finally, when a model fails validation, the **invalid** event is triggered. Using the **on** function, an event handler can be attached to the model to handle the **invalid** event. A common event handler would hook into the UI and display the validation error messages.

Exercise

Step 1. With a text editor, open the file **index.js** file from folder **exercises/models/js**.

Step 2. Create a new model named **Widget** that inherits from **Backbone.Model**.

Step 3. Using the **defaults** property, to set default values for the model attributes named **name**, **description**, **size** and **color**. The default value for each attribute is an empty string.

Step 4. Add a function to the Model named **getLongDescription**. The function should return a value of the four attributes concatenated together with a space between each value.

Step 5. Instantiate a new model, and assign it to a variable named **aWidget**.

Step 6. Setup an event on the new model to output the new **color** value to the console each time the **color** attribute is modified.

Step 7. Set each attribute to a realistic value. When the color attribute is set make sure the new value is outputted to the console.

Step 8. Call the **getLongDescription** function on the model and output the value to the console.

Step 9. Open the file **exercises/models/index.html** from the file system with the web browser of your choice. Open the JavaScript Console (it's part of the Developer Tools for the browser). Reload the page, and observe the output to the console.

Congrats you have completed the exercise!

Review Questions

1. What is the purpose of Backbone models?
2. What is the difference between the initialize function and the constructor function?
3. What two methods are used retrieve and change the values of attributes?
4. What method is used to register event handlers for model events?

Backbone Collections

Objectives

1. Understand the purpose of Backbone Collections
2. Learn how to create a collection that inherits from Backbone.Collection
3. Explore how to add a model to and remove a model from a collections
4. Explore how to iterate over collections

Introduction

The Backbone.Collection object provides the functionality needed to manage many model objects as a list or collection.

Create a New Collection

To create a new collection, the Backbone.Collection object needs to be extended.

```
var People = Backbone.Collection.extend({
    model: Person
});

var myPeople = new People();
```

The **extend** function configures the **People** object to prototypally inherit from the Backbone.Collection object. To use the People collection, a new instance of the People collection object needs to be instantiated with the **new** operator.

Adding Items to a Collection

An empty collection is of little value, so models must be created, and added to the collection. To add a model to the collection, the **add** function is called.

```
myPeople.add(new Person({
    firstName: "John",
    lastName: "Doe"
}));

myPeople.add(new Person({
    firstName: "Jane",
    lastName: "Doe"
}));
```

Observe how the **add** function is used to add a model to the collection.

Removing Items from a Collection

In addition to adding items to a collection, items can be removed from a collection as well. The function to remove an item is named **remove**.

```
var aPerson = new Person({
    firstName: "John",
    lastName: "Doe"
});

myPeople.add(aPerson);

myPeople.remove(aPerson);
```

Observe, how the item is removed from the collection based upon the object's reference.

Retrieving an Item from a Collection

To retrieve an item from a collection using an **id**, the **get** function can be used.

```
var aPerson = myPeople.get(personId);
```

Observe how the model id is passed into the **get** function and the model is returned.

Iterating over a Collection

A common operation for collections is to iterate over the collection, and process each item.

```
myPeople.each(function(person) {
    console.log(person.get("firstName"));
});
```

Using the **each** function provided by the Underscore library (as exposed by the Backbone.Collection object), the code can iterate over the collection, passing each item in the collection into a function, where some kind of logic can be applied to the item.

Exercise

Step 1. With a text editor, open the file **index.js** file from folder **exercises/collections/js**.

Step 2. Copy the Widget model from the previous exercise into the **index.js** file. Add an **id** attribute to the model if one does not exist.

Step 3. Create a new collection object named Widgets that inherits from the Backbone.Collection object.

Step 4. Configure the collection to the use the Widget model.

Step 5. Instantiate a new Widgets collection named **theWidgets**.

Step 5. Create five new Widget objects and add them to the **theWidgets** collection. For each Widget, specify reasonable values.

Step 6. Iterate over the **theWidgets** collection, calling the **getLongDescription** function, outputting the value to the console.

Step 7. Remove the third Widget object from the collection.

Step 8. Iterate over the collection of Widget models again, calling the **getLongDescription** function, outputting the value to the console.

Step 9. Open the file **exercises/collections/index.html** from the file system with the web browser of your choice. Open the JavaScript Console (it's part of the Developer Tools for the browser). Reload the page, and observe the output to the console.

Congrats you have completed the exercise!

Review Questions

1. What is the purpose of Backbone collections?
2. What function is used to define a new Backbone collection object?
3. What function is used to iterate over a collection?

Views

Objectives

1. Understand the purpose of Backbone Views
2. Learn how to create a view that inherits from Backbone.View
3. Explore how to add a template engine and events to a view
4. Explore how to instantiate and configure a view

Introduction

Backbone provides the **Backbone.View** object to define a structure for providing the user interface of the application's model and collection data, including wiring up and handling user interface events such as a button click. Unlike other JavaScript libraries and frameworks, Backbone's View object is not opinionated about which template system to use or whether to support two-way data binding. Backbone Views allow the developer to choose what is best for the application.

Create an Underscore.js Template

The default template engine use by Backbone.js is the Underscore.js template engine provided by Underscore.js. The Underscore.js template engine is a string based template engine that provides very few features. However, it is an extremely fast, and easy to use template engine.

The Underscore **template** function compiles the template producing a template function. A data object is then passed into the template function and results in a string of HTML filled in with actual data. This string is then added to the DOM.

Underscore template commands are wrapped in one of three sets of command brackets:

<%= someVariableOrExpressionToOutput %> - this will output an interpolated non-escaped HTML string value

<%- someVariableOrExpressionToOutput -%> - this will output an interpolated escaped HTML string value

<% javascriptCodeToExecute %> - this executes some JavaScript, and can be used with multi-line constructs like **if** statements and **for** loops

```
var
    data = { firstName: "Bob", lastName: "Smith" };
    htmlTemplate = "<%= firstName %> <%= lastName %>";
    templateFn = _.template(htmlTemplate);
    htmlOutput = templateFn(data);
```

The variable **htmlOutput** would contain the final HTML with both the data applied to template.

Create a Handlebars Template

One of the more common template engine libraries is called Handlebars. In order to use Handlebars, the Handlebars JavaScript library needs to be included as described in the course introduction.

The following **SCRIPT** element contains the definition of a Handlebars template. Observe the **id** and **type** attributes on the **SCRIPT** element. The **id** attribute will be used to load the template, and the **type** attribute informs the browsers to **NOT** execute the **SCRIPT** block as JavaScript.

```
<script id="person-view" type="text/x-handlebars-template">
    <div>
        <h2>View Person</h2>
        <div>
            <label>First Name:</label>
            <span>{{firstName}}</span>
        </div>
        <div>
            <label>Last Name:</label>
            <span>{{lastName}}</span>
        </div>
        <div>
            <button class="button edit">Edit</button>
        </div>
    </div>
</script>
```

The following code takes the **person-view** template and applies the Handlebars template engine to it. The template process goes through five steps. First, the template is retrieved. Second, the template is compiled. The result of the compilation is a function. Third, the data to be bound to the template is prepared. Next, the data model is passed into the template function. Finally, the result of the template function is the HTML to be added to the DOM.

```
// HTML cannot be added to the body until the DOM is loaded
window.addEventListener("DOMContentLoaded", function() {

  var
    // Step 1: Get the Template HTML Source
    source = document.getElementById("person-view").innerHTML,

    // Step 2: Compile the Template
    template = Handlebars.compile(source),

    // Step 3: Setup Data Model
    context = { firstName: "Bob", lastName: "Jones" },

    // Step 4: Apply Data Model to the Template
    html = template(context);

  // Step 5: Replace Body Content with Data-Bound Template
  document.body.innerHTML = html;

});
```

Create a View

To create a new view, Backbone.View needs to be extended as shown below. This will allow the new view to prototypally inherit from the Backbone.View object.

```
var PersonView = Backbone.View.extend({
  tagName: "div"
});
```

The **tagName** property defines the HTML element tag that will wrap the view when it is rendered. If no tag name is specified, the default value is a **div** element.

Add Template Engine to a View

Once a new view object has been defined, a template engine needs to be added to it. As mentioned earlier, Backbone gives you full control over which template engine the application uses. In this example, the Handlebars template engine is being used.

```
var PersonView = Backbone.View.extend({

  tagName: "div",

  id: "person-view",

  initialize: function() {
    var source = document.getElementById(this.id).innerHTML;
    this.template = Handlebars.compile(source);
    this.render();
  },

  render: function() {
    var html = this.template(this.model.attributes);
    this.$el.html(html);
  }

});
```

Observe the **initialize** and **render** functions added to the view object. The **initialize** function is being used to load the template source and compile it into a template function. Then the initialize function fires off the render function.

The render function uses the **template** function to pass in the **model** which then generates the HTML. The HTML is added to the DOM through a call to the **html** function of the jQuery wrapped DOM element accessed through the property **$el**. The **$el** property is set through the **el** property where the raw DOM element is set when instantiating a new view. The setting of the **el** property on the view will be covered in a few sections from now.

Add Events to a View

In addition to displaying data bound to a template, views are used to handle user interface events such as button clicks. The **events** option is an object map that uses a property to determine which event will be listened for and a selector to determine on which DOM elements will be listening. The value of the object map property is the name of the function object property defined on the view object. Once the rendering process is complete, the events are applied to the live DOM elements.

```
var PersonView = Backbone.View.extend({

  tagName: "div",

  events: {
    "click .button.edit": "edit"
  },

  id: "person-view",

  initialize: function() {
    var source = document.getElementById(this.id).innerHTML;
    this.template = Handlebars.compile(source);
    this.render();
  },

  render: function() {
    var html = this.template(this.model.attributes);
    this.$el.html(html);
  },

  edit: function() {
    console.dir(arguments);
  }

});
```

Observe the **edit** button event configuration. In the template there is **button** element with the classes, **button** and **edit**, applied to it. When the view runs, the DOM selector specified in the events object will be used to select the button element and the **edit** function will be bound to its **click** event.

Events configured with the events hash can be activated and deactivated with the **delegateEvents** and **undelegateEvents** functions. If no events hash is passed into the **delegateEvents**, then the **events** option hash on the view is used.

Using the View

To use the view, a new instance of it must be created using the **new** operator. An object is passed into the constructor function to set various options. In this case, the **el** option and the **model** option are being set. The **el** option is the DOM element to which the view content is going to appended as a child in the

DOM. The **model** option specifies the model the view is going to be bound to.

```
// HTML cannot be added to the body until the DOM is loaded
window.addEventListener("DOMContentLoaded", function() {

  var personModel = new PersonModel();
  personModel.set("firstName", "Eric");
  personModel.set("lastName", "Greene");

  var personView = new PersonView({
    el: $(document.body),
    model: personModel
  });

});
```

Because the **render** method is wired up to the **initialize** function (see earlier code samples), once the view is instantiated, the template will be compiled, the model data will be passed to the template function, and the resulting HTML string will be added to the DOM. After the render function executes, the events will be wired up to the specified DOM elements for the events they are specified for.

Destroying the View

When a view needs to be disposed, a number of actions are needed; otherwise, memory leaks can occur. First, events bound to the view need to be cleaned up. Event bindings that are not removed will result in web browser memory leaks. These leaks occur because the DOM and JavaScript represent two different contexts in the web browser. Each has their own memory space, their own garbage collector and are truly different subsystems. Many developers think that DOM objects are JavaScript objects in memory, but they are not. DOM objects are C++ objects, while the DOM API allows web developers to manipulate these C++ objects through a JavaScript API.

The **View** object provides a remove function that will remove the view's DOM tree from the DOM, and it cleans up all events registered with the **events** option on the view object. Any events not registered with the **events** option will need to manually cleaned up.

Review the code to remove a view:

```
personView.remove();
```

Unfortunately, the **remove** method does not trigger a **remove** Backbone event that the developer can bind to execute custom view removal logic. Therefore, for a view with custom DOM event bindings (not using the **events** options) or for views requiring additional clean up logic, a good solution is to override the remove method on view prototype. The following code demonstrates how to implement such logic.

```
var CustomView = Backbone.View.extend({
    remove: function(){

        // Custom removal code goes here...

        Backbone.View.prototype.remove.apply(this, arguments);
    };
});
```

The view framework of Backbone.js provides a solid foundation for managing views, while offering a tremendous amount of flexibility for how the details of the view are actually implemented.

Exercise

Step 1. With a text editor, open the file **index.js** file from folder **exercises/views/js**.

Step 2. Create a new view named **WidgetView** that inherits from **Backbone.View**.

Step 3. Set the tag name for the view to **div**.

Step 4. Create an **initialize** function to load the template with an **id** of **widget-view**.

Step 5. Compile the template within the initialize function assigning it to a view property named **template**, then call the **render** function.

Step 6. Implement the render function to call the view's **template** function, passing in the model object. Append the HTML returned by the **template** function to DOM.

Step 7. Copy the Widget model from the Backbone Model exercise. Instantiate a new Widget model, assign reasonable values to its attributes.

Step 8. Instantiate a new Widget View, passing in the Widget model and the **document.body** for DOM element the view will be appended to.

Step 9. Open the file **exercises/views/index.html** from the file system with the web browser of your choice. The Widget View will appear populated with widget data. If the Widget View does not appear, review your code, and make any needed adjustments.

Congrats you have completed the exercise!

Review Questions

1. What are Backbone Views?
2. Does Backbone require that the application use a particular template engine?

3. What view object function is used to generate the HTML for the template?
4. Can Backbone Views be configured to handle events?

Routing

Objectives

1. Understand the purpose of Backbone Routing
2. Learn how to create a router that inherits from Backbone.Router
3. Explore how to add a routes, and functions to handle those routes

Introduction

Routing is the mechanism by which a web application decides which view to show based upon some kind of condition. The condition typically used by web applications to determine routes is the current URL or some kind of internal state. Additionally, routing can be done on the web server or on the web browser client. Server-side frameworks such as ASP.Net MVC or Java's Spring MVC offer server side routing mechanisms. Backbone offers URL-based client side routing through the Backbone.Router object.

Creating a Router

As with the earlier examples, to create a new router object, the Backbone.Router object must be extended. Additionally, an initialize function has been provided to pass along any options that passed into the router.

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

  initialize: function(options) {
    this.options = options;
  }

});
```

Defining Routes

Once a new router object has been created to prototypally inherit from the Backbone.Router object, then the routes need to be defined. The routes are defined on the **routes** property on the router object. The **routes** property is an object map where the property name is the URL route pattern and the property value is the name of the function on the router object that will be called when changing to a route.

```
PersonRouter = Backbone.Router.extend({
    routes: {
      'person/:id': 'viewPerson',
      'person/:id/edit': 'editPerson',
      'people': 'showPerson',
      '': 'showPerson'
    },

    initialize: function(options) {
      this.options = options;
    }

  });
```

In the URL pattern, observe the **:id** portion of the pattern. This represents the portion of the URL that will be passed in as an **id** parameter to the route function.

Once the routes are defined, the functions named in the **routes** object map needed to be defined on the **PersonRouter** object.

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

    routes: {
      'person/:id': 'viewPerson',
      'person/:id/edit': 'editPerson',
      'people': 'showPeople',
      '': 'showPerson'
    },

    viewPerson: function(id) {
      if (this.currentView) {
        this.currentView.undelegateEvents();
      }
      this.currentView = new ViewPersonView({
        el: this.options.el,
        model: people.get(id),
        router: this
      });
      this.currentView.render();
    },

    editPerson: function(id) {
      if (this.currentView) {
        this.currentView.undelegateEvents();
      }
      this.currentView = new EditPersonView({
        el: this.options.el,
        model: people.get(id),
        router: this
      });
      this.currentView.render();
    },
    showPeople: function() {
      if (this.currentView) {
        this.currentView.undelegateEvents();
      }
      this.currentView = new PeopleView({
        el: this.options.el,
        collection: people,
        router: this
      });
      this.currentView.render();
    },

    initialize: function(options) {
      this.options = options;
    }

  });
```

Observe how each route function instantiates a new view object, passing the DOM element to append the view to as well as the model data for the view.

For the routing system to begin working on the web page, a new instance of the router object needs to instantiated. Additionally, Backbone needs to be told to start paying attention to the URL so it can change routes. The code to do this is as follows.

```
    var router = new PersonRouter({ el: $("body") });
    Backbone.history.start({pushState: false});
```

It is recommended to place this code inside of a function that fires when the DOM is loaded. This is easily done with jQuery's **$(document).ready** function.

```
  $(document).ready(function() {

      var router = new WidgetsRouter({ el: $("body") });
      Backbone.history.start({pushState: false});

  });
```

When the DOM is fully loaded, the router object will be created and Backbone will be told to start paying attention to the URL.

Viewing a Route

To view a route from using the route definitions from above the following URLs would be used.

http://www.mysite.com/ - displays the list of people

http://www.mysite.com/people - displays the list of people

http://www.mysite.com/person/1 - displays the person with an id of 1

http://www.mysite.com/person/1/edit - display an edit form for the person with an id of 1

Exercise

Step 1. With a text editor, open the file **index.js** file from folder **exercises/routes/js**.

Step 2. Create a new router object named **WidgetsRouter** that inherits from **Backbone.Router**.

Step 3. Add a route property with the following defined routes.

1st Route: **widget/:id** will be handled by **viewWidget** 2nd Route: **widget/:id/edit** will be handled by **editWidget** 3rd Route: **widgets** will be handled by **viewWidgets** 4th Route: **[empty string]** will be handled by **viewWidgets**

Step 4. Implement the function **viewWidget** with the **ViewWidgetView** view object to display the view with the specified model. Note: **ViewWidgetView** is already defined within the exercise files. You do not need to implement the view object itself.

Step 5. Implement the function **editWidget** with the **EditWidgetView** view object to display the view with the specified model. Note: **EditWidgetView** is already defined within the exercise files. You do not need to implement the view object itself.

Step 6. Implement the function **viewWidgets** with the **WidgetsView** view object to display the view with the specified model. Note: **WidgetsView** is already defined within the exercise files. You do not need to implement the view object itself.

Step 7. Instantiate a new **WidgetsRouter** object, passing the a jQuery wrapped **BODY** element as the value for **el**. Be sure to run this code after the DOM has been loaded.

Step 8. Open the file **exercises/routes/index.html** from the file system with the web browser of your choice. Using the links on the web page, navigate between the web pages.

Congrats you have completed the exercise!

Review Questions

1. What is the purpose of the Backbone.Router object?
2. Does Backbone.Routers use state based routing or URL based routing?
3. What is the format for parameters that are extracted from the URL?

4. Is Backbone routing server-side or client-side routing?

Syncing Models and Collections

Objectives

1. Understand JavaScript's Single-Threaded Model and Asynchronous Programming
2. Explore the XMLHttpRequest Object and the concept of AJAX
3. Utilize the model and collection AJAX synchronization capabilities provided by Backbone.js

Introduction

JavaScript is a single-threaded, event-driven programming language. It's execution model is event driven whereby jobs (aka tasks) are executed on an event loop. Before being pushed on to the event loop, jobs are stored in one of several queues with varying scheduling priorities. While some kinds of jobs are assigned to higher priority queues than other kinds of jobs, once a job has been pushed on to the event loop, JavaScript will execute that job without interruption until it is complete. Completing the job without interruption means that no matter the length of time it takes to execute the job, JavaScript will execute it until it is done. This means that even for a long running job, JavaScript will not swap tasks to complete other jobs. Tying in JavaScript'a single-threaded natures, means there is no thread switching either.

JavaScript is described as event-driven because it is driven by jobs that are executed in response to some kind of event. Common events in a web application include button clicks, dragging and dropping as well as AJAX calls using the XMLHttpRequest object. When these events occur, the event handler registered for the event is the job pushed on to the event loop. The function, and all of the functions it will execute, represent the complete job that will be executed without interruption.

While JavaScript is single-threaded, it is surrounded by an environment that is not single-threaded. Web browsers are made of many components such as the DOM, networking, etc... These other components of the web browser are usually programmed in C++ (or another multi-threaded language) and are capable of more sophisticated execution models. JavaScript interacts with these components in a single-theaded asynchronous manner, while the components themselves operate in a multi-threaded manner. To better understand this model, it helps to think of JavaScript as glue language for C++ modules. To enable JavaScript efficiently operate asynchronously, two programming patterns are utilized: callback functions and closures. Callback functions allow JavaScript to make a function call asynchronously.

To illustrate, JavaScript code calls a API function provided by the browser, passing a callback function to the API call. The API function is really a C++ web browser component that will perform some kind of asynchronous operation. After the call is made, JavaScript immediately moves onto the next instruction on its stack, allowing the C++ code to executes its operations on another thread. Once the C++ web browser component completes its operations, it will call the push the callback function on to the event loop passing in the result data from its operations.

This pattern allows JavaScript to keep executing while other components do the heavy lifting of executing CPU intensive and IO intensive operations. Finally, in order for the callback function to have a reference to the original variable used in the original JavaScript task that requested the asynchronous API function call, closures are used. To see a great example of this let's consider an AJAX call made with jQuery's $.ajax function.

jQuery AJAX calls

Most web developers have experience making AJAX calls with jQuery's **ajax** function. While jQuery provides a promise-based interface, almost all developers are familiar with the older style **success** and **error** functions. From the developer perspective, jQuery's **ajax** function makes the AJAX call, then when the response from the server is received the **success** function is invoked passing in the data received from the server. Or if an error occurred (as determined by the response code from the server), the error function is invoked passing in the error information. Review the code below:

```
$.ajax({
    method: "POST",
    url: "/api/widgets",
    contentType: "application/json",
    success: function(data, jqXHR, textStatus) {
        // process the data
    },
    error: function(jqXHR, textStatus, errorThrown) {
        // handle the error
    }
});
```

Many developers intuitively think of this code as being synchronous in nature even if they know its asynchronous. Behind the scenes the following takes place. The call to jQuery's **ajax** function causes the browser to instantiate an XHR object. The options needed to make the request are passed to the XHR object, and it's send function is invoked to make the request. The send function immediately returns control back to JavaScript and the rest of the JavaScript job on the event loop is completed.

While JavaScript continues executing jobs on its single-thread, the web browser's C++ components execute the actual network call on another thread. When the response is received, the XHR object triggers the success/error function passed in from the original jQuery **ajax** function. The function is triggered by pushing it on the event loop, where it is executed as a separate job from the original call. Through the use of closures, the original function can reference the original variables that were available in the scope of the original jQuery **ajax** call.

```
function getWidget(widgetId) {

    // the call to the AJAX function occurs within one job, and the results
    // will processed in a future job
    $.ajax({
        method: "GET",
        url: "/api/widgets/" + encodeURIComponent(widgetId),
        contentType: "application/json",
        success: function(data, jqXHR, textStatus) {

            // even though this function is running is a completely
            // different job on the event loop, the widgetId variable
            // is available via closure
            console.log(widgetId);

        },
        error: function(jqXHR, textStatus, errorThrown) {

        }
    });

}
```

JavaScript's asynchronous programming ability is actually quite powerful when used for non-blocking I/O purposes such as AJAX calls.

Backbone Sync

Backbone encapsulates the XHR mechanisms for keeping objects synchronized with the server through its **sync** function. The synchronizing process uses jQuery's low-level **ajax** function to make AJAX calls through the browser's **XMLHttpRequest (XHR)** object. Backbone expects the server to implement the standard REST service pattern corresponding to the models and collections that are being synchronized (this standard pattern can be overridden, AJAX and sync function customizations are described later).

The pattern is as follows:

- Get All Models in a Collection
    - HTTP Method: GET
    - URL Pattern: /collection-name
    - Request Body: None
- Get One Model in a Collection
    - HTTP Method: GET
        - URL Pattern: /collection-name/model-id
        - Request Body: None
- Create a New Model
    - HTTP Method: POST
    - URL Pattern: /collection-name
    - Request Body: JSON data containing new model

- Update the Whole Existing Model
    - HTTP Method: PUT
    - URL Pattern: /collection-name/model-id
    - Request Body: JSON data contains the updated model, the id in the model is ignore, and the id from URL is used to determine which model should be updated
- Update Some Fields of an Existing Model
    - HTTP Method: PATCH
        - URL Pattern: /collection-name/model-id
        - Request Body: JSON data contains the fields of the model to be updated (id should not be included), and the id from URL is used to determine which model should be updated
- Delete One Model
    - HTTP Method: DELETE
        - URL Pattern: /collection-name/model-id
        - Request Body: None

The **sync** function exposes access to the REST service operations through the following function signature:

Backbone.sync(method, model [, options])

The following values for **method** are supported:

- "create" - POST request which creates a new model
- "read" - GET request which reads a model or a collection of models
- "update" - PUT or PATCH request which updates a model
- "delete" - DELETE request which deletes the model

The **model** is the model to be saved, or the collection to be populated.

The **options** are used to specify **success** and **error** functions, as well as jQuery AJAX options.

Typically, developers do not call the **sync** function directly; instead, there are specific sync helper methods on models and collections that when called will invoke the **sync** function to perform the various AJAX operations.

Sync Functions on the Models and Collections

As described in the previous section, Backbone models and collections expose the **Backbone.sync** function for synchronizing models and collections with the server. While calling the **sync** function is an acceptable option, Backbone provides helper functions to make performing the commonly used CRUD operations easier. For models, the three functions **fetch**, **save**, and **destroy** are use to retrieve, create/update and delete models. For collections, there are two functions: **fetch** and **create**.

Models

fetch - Using the **id** of the model, fills the attributes of the model from the current values stored on the server. The **fetch** function takes a single options argument which is used to specify **success** and **error** functions. If the data retrieved from the server does not match the current values on the attributes of the model, then a Backbone **change** event is triggered.

save - If the **isNew** property of the model is true, then a new object will be created on the server; otherwise, the **id** attribute will used to update an existing model. The **save** function accepts two arguments, an **attributes** parameter and an **options** parameter. The **attributes** parameter allows the assignment of new values to the model's attributes before the create/update operation is executed. If new attribute values are supplied, a Backbone **change** event will be triggered, the request will be made (triggering a Backbone **request** event), and a Backbone **sync** event will be fired once the request has successfully completed.

When specifying new attribute values, if the model has validation logic, and the validation logic fails, then the save operation will be cancelled.

In addition to the attributes parameter, a number of options can be specified as well. If the save operation should only update the changed attributes on the server, not all of the attributes, the **patch** option can be set to **true** on the options object. When specifying new attribute values, if the model should not be updated until AJAX request has be completed then pass the option **wait** set to true on the options object. Finally, **success** and **error** functions can be set on the options object to handle the response.

Also, any additional jQuery AJAX options can be specified on the object and passed through to jQuery's AJAX function. Passing jQuery options should be limited as it could introduce unnecessary coupling between the application code and the jQuery library.

destroy - Deletes the model from the server, and removes it from any collections of which it is a member. Destroy triggers a Backbone **destroy** event which bubbles up to the collections. Similar to **fetch** and **save**, the Backbone **request** event fires when the request is made, and upon successful completion the Backbone **sync** event fires. Similar to the **wait** option for attribute updates with the **save** function, the **wait** option can be used to delay the local destruction of the model until the model has received a response on deletion from the server.

Collections

fetch - The **fetch** function retrieves the collection from the server. The only parameter the **fetch** function accepts is the **options** argument. For **success** and **error** functions, set the success and error properties on the **options** object.

create - The **create** function create a new model in the collection, and saves it to the server.

Success and Error Functions

For all operations, **success** and **error** functions can be specified. While its beyond the scope of this manual, these **success** and **error** functions can be easily wrapped in promises to promisify the Backbone sync operations. Both the **success** and **error** function are passed the following arguments:

Model Sync Operations

```
function success(model, response, options) {}

function error(model, response, options) {}
```

Collection Sync Operations

```
function success(collection, response, options) {}

function error(collection, response, options) {}
```

The **model** and **collection** arguments are the parsed model or collection from the JSON response. The **response** argument is the original JSON response. The **options** are the options passed into the **Backbone.ajax** function and includes a reference to the **XHR** object used for the request.

Customizing AJAX Requests

Backbone exposes the function used to make AJAX calls through the **Backbone.ajax** property. The default implementation of this function used jQuery's low-level **ajax** function to make the AJAX requests. However, this default implementation can be overridden as needed. For web applications which do not use jQuery, this function can be replaced with an implementation that uses another AJAX library or even direct XHR access. The following code demonstrates replacing the default **ajax** function with a non-jQuery version using the built-in XHR object.

```
Backbone.ajax = function(options) {

    var xhr = new XMLHttpRequest();

    xhr.onreadystatechange = function() {
        if (xhr.status !== 200 && xhr.readyState > 1) {
            options.error(xhr);
        }
        if (xhr.readyState == 4) {
            options.success(xhr);
        }
    };

    xhr.open(options.type, options.url);
    xhr.setRequestHeader("Content-Type", options.contentType);
    xhr.send(options.data);
}
```

In addition to overriding the **ajax** function, the **sync** function itself can be overridden. Typically, the **sync** function is overridden when the core AJAX code provider is sufficient, but the application needs pre/post-processings hooks into the AJAX request or responses. Such modifications include adding custom headers, transforming request/response data, logging errors, etc... Generally, the **sync** method does not need to be completely replaced, it just needs to be intercepted as shown below. A base model for the application can be created, then that base model can "override" the original **sync** function. Then within the override function, the original **sync** method is called using **apply** after custom processing has been completed. Review the following code:

```
var BaseModel = Backbone.Model.extend({
    sync: function() {

        // configure a before send function
        if (!arguments[2]) {
            arguments[2] = {};
        }
        arguments[2].beforeSend = function(xhr) {
            // add a custom header
            xhr.setRequestHeader("X-CSRF-Token", "5mwH9VTi-WGHWxVYMyDxVKMlmsNa3adp_r
        };

        Backbone.sync.apply(this, arguments).then(function() {
            // retrieve the new CSRF token from the response
            window.csrfToken = arguments[2].getResponseHeader("X-CSRF-Token");
        });
    }
});
```

The above code sample demonstrate the true flexibility and extensibility of JavaScript, its many libraries and especially Backbone. Finally, it is worth mentioning that when using jQuery for AJAX calls many of the same customizations described in this section can be accomplished using the global AJAX handlers provided by jQuery. However, there are several down sides to using these global handlers. First, these handlers will apply to all AJAX calls made with jQuery not just the Backbone triggered calls. Secondly, the global handlers will not have direct access to the models and collections making the calls. Finally, using the handlers will tightly couple the application to jQuery for performing AJAX calls.

Additional Options on the Sync Object

There are two additional options available on the **Backbone** object related to synchronization that are used when working with legacy web servers or very restrictive networks.

emulateHTTP - Some legacy web servers are not configured to accept HTTP PUT, PATCH and DELETE requests. Even some very restrictive networks will filter out such requests. To work around this scenario, this option can be set to **true** (default is **false**), and Backbone will use a POST request for these requests and set an additional header named **HTTP-Method-Override** which will list the appropriate HTTP method. The REST services can then use the value of this header to execute the appropriate operation.

emulateJSON - Some legacy web servers and application platforms are not able to handle JSON. For these situations, this option can be set to **true** (default is **false**) and the model data in the request body will be submitted as a url-encoded form posting with the model data assigned to a form variable named **model**.

Review Questions

1. Is JavaScript multithreaded? Is asynchronous programming primarily intended for single-threaded environments or multi-threaded environments?
2. What browser object is used to perform AJAX operations?
3. What which JavaScript library does Backbone.js use by default to perform AJAX operations?
4. Can the sync method provided by Backbone.js be replaced with a custom method? What is one instance where this might be needed?

Conclusion

Backbone is a great JavaScript library for building single web page applications. It provides the basic functionality needed for managing data, building views and routing users from part of the application to the other.