Views

ROHAN RAJORE

Backbone Views

- When speaking of Backbone.View, we are referring to the JavaScript objects that are created in a Backbone application to organize the code into logical views.
- This means that although a view can update the HTML that is present in a view, usually with a templating library, the view will listen for changes in the model and render the changes on a designated section of your HTML page.

Creating Backbone Views

- Creating your own view definition is done through the Backbone.View.extend() function. Just as with Backbone.Model and Backbone.Collection, you can pass an I nitialize function that will get invoked on construction of a new instance.

```
//Define the Library View
LibraryView = Backbone.View.extend({
initialize: function(){
console.log('View created');
},
});
```

- To create a new instance of your view, simply use the new operator.

```
//Create an instance of the view
varmyView = new LibraryView();
```

Creating Backbone Views

- You can pass additional options when creating the view, such as the model that the view is being attached to and the element that the view should be attached to.
- In the following code snippet, an instance of a Book model is passed through to the LibaryView, which can be accessed in the initialize function of the view:

```
varthisBook = new Book({name : 'Beginning Backbone', author: 'James
Sugrue'});
//Create an instance of the view
varmyView = new LibraryView({model: thisBook});
```

Binding to the Physical View

- To bind your View class to an HTML page, you will need to utilize the el attribute, which is at the center of any view.
- el is the reference to the DOM element on your HTML page that is associated with the view.
- The DOM element is available at all times, whether the view has been rendered or not. Because of this, the entire view can be created before it is rendered, at which point the element is inserted into the DOM at once, with as few repaints as possible.
- There are two ways to use el: by referencing an existing DOM element or by creating a new one.

Linking el of the view to an existing DOM element

- By passing an el attribute to the constructor, you are telling the view which DOM element itshould attach to.
- For example, if the HTML page had a div created to contain the library view contents, you would want to pass a reference to that element when constructing the view.

```
<body>
<div id="myLibraryViewSection"></div>
```

- The reference is made using standard CSS selectors that use #<element_id> or .<element_class>.

```
//Create an instance of the view with a model and an element
varmyView = new LibraryView({
model: thisBook,
el: '#myLibraryViewSection'
});
```

Creating el for the view dynamically

- You can also create the el for the view dynamically by passing a number of properties to the view when constructing it.
- tagName: The name of the HTML element to use for the view. If none is specified, the value of tagNamewill default to div.
- className: The CSS class that will be used to render this element. This property is optional. You can specify a number of classes for the element, passing them through as space-separated values.
- id: The ID to assign to the element. This property is optional.
- attributes: Additional attributes to assign to the element, such as data- attributes in name-value pairs.

Creating el for the view dynamically

```
//Create a view that will build its own DOM element
varmyNewView = new LibraryView({model: thisBook,
tagName: 'ul',
className: 'libraryview',
id: 'library',
attributes: {'data-date': new Date()}
});
console.log(myNewView.el);
```

- The result of myNewView.el will now be as follows:

```
<ul id='library' class='libraryview' data-date='Mon Aug 05 2013 13:54:20
GMT+0100 (IST)'></ul>
```

- If none of these properties is passed through, the el is created as an empty div element.

Rendering Content

- To get your view content onto a page, you will need to override the render function. Left unimplemented, the content of the view will never be rendered. To have the view render automatically on construction, you could call the render function from the view's initialize function.

```
//Define the Library View
LibraryView = Backbone.View.extend({
   initialize: function(){
      this.render();
   },
   render: function(){
      this.$el.html('Hello Library');
      return this;
   }
});
```

Rendering Content

- Of course, a more useful render function would make use of the model or collection that is present in the view.

```
render: function(){
    this.$el.html('Book Name: ' + this.model.get('name'));
    return this;
}
```

- Note how the render function has a return this statement at the end. This is a convention that is encouraged by Backbone to allow chained calls.

Passing Collection to a View

- Note that rather than using a single model in our rendering, we could also pass a collection to the view when creating a new instance.

```
varmyLibraryView = new LibraryView({
collection: myLibrary,
el: '#myLibraryViewSection'
});
```

- The render method could now deal with the entire collection of models.

```
render: function(){
   for(vari =0; i<this.collection.size(); i++){
      this.$el.append('<li>Book Name: ' + this.collection.at(i).get('name')
+ '</li>');
   }
   return this;
}
```

Removing Views from DOM

- it is possible to remove the view entirely from the DOM. This will also result in all event listeners being disposed of.

```
myLibraryView.remove();
```

Finding Elements within the view

- It is common to need to find nested elements within your view and run other jQuery functions within the view’sscope. Backbone provides an additional $el property that functions as a shorthand for $(view.el).
- For example, to find a subelement within your view, you can use the following code. This will find the element with the ID Beginning Backbone if it exists within the view.

```
$el.find('#Beginning Backbone');
```

- Similarly, Backbone provides a $(<selector>) function as shorthand for (view.el).find(<selector>), making any code that deals with subelements more elegant and readable.

```
$('#Beginning Backbone');
```

- Both snippets of the previous code achieve the same result: finding an element with the ID of myelement within the scope of the current view.

View Events

- Now that the view is rendering, you will probably want to add some events to make it interactive. Once again, this is made simple in Backbone with the ability to specify a hash of events in the view definition.
- The events listed in the hash will use jQuery’son function to provide callbacks for DOM events with the view.
- The format {'event selector': 'callback'} is used to define the events that will be handled.

```
LibraryView = Backbone.View.extend({
    events: {
        'click #book' : 'alertBook'
    }
});
```

View Events

- If no selector is defined, the event will be bound to this.el, the root element of the view.
- For example, to add a click handler to the entire view, you could simply use this:

```
events: {
'click : 'alertBook'
},
```

View Events: delegateEvents

- If you need to define events individually for each instance of the view or if a view has different modes of operation, you can use delegateEvents(). This function is called from the View constructor automatically, so you don't need to call it when your events hash has been defined. Calling delegateEvents will remove any existing callbacks and use the events hash provided to create a new set of event handlers.
- The following would change the listener for the alert from a click to a mouseover:

```
myLibraryView.delegateEvents( {
'mouseover #book ' : 'alertBook'
});
```

View Events: undelegateEvents

- You can remove all events by passing an empty events hash to delegateEvents or by calling the undelegateEvents function. By adding a line such as the following to your view, you can remove all the events handled in myLibaryView.
- Type the following line into your Chrome Developer Tools console, and notice how the view no longer responds to any events you have created handlers for.

```
myLibraryView.undelegateEvents();
```

Self and this

- A lot of the code that you will see in Backbone applications will have functions where a reference to the original this object is saved as self.

```
var self = this;
```

- This is done to maintain a reference to the original this, even when the context changes. You’ll see this happen a lot in closures in your code, especially in any event handlers you create. Using this approach, the render function would change to use the self reference in place of this.

```
render: function(){
   var self = this;
   for(vari =0; i<this.collection.size(); i++){
      self.$el.append('<li id="book">Book Name: ' +
      self.collection.at(i).get('name') + '</li>');
   }
   return self;
},
```