Collections

ROHAN RAJORE

Backbone Collections

- In the previous section we focused on the single models, but usually Backbone applications use Backbone.Collection to provide ordered sets of models. This has some useful side effects, such as being able to fetch an entire collection from a back-end server and listening for events across any of the models in a collection.
- When defining a collection, you will always pass through the model that is being contained. To follow on with the book example used in the Backbone.Model section, we will define our collection of books as a library.

```
//Define a collection based on book
var Library = Backbone.Collection.extend({model: Book});
```

Backbone Collections

- Just as with Model, an initialize method can be provided in your definition of the collection, which can be invoked on construction of a new instance of the collection. This can be useful to set up event listeners for the collection.

```
var Library = Backbone.Collection.extend({model: Book,
initialize: function(){
console.log('Creating a new library collection');
}
});
```

- When a new Library object is created, you'll see the console.log statement from the initialize function print out to the console.

```
var myLibrary = new Library();
```

Constructors

- When creating an instance of the collection, you can pass through an array of model objects to populate it with some initial content.

```
var bookOne = new Book({name: 'Beginning Backbone', author: 'James
Sugrue'});
var bookTwo = new Book({name: 'Pro Javascript Design Patterns',
author:'Dustin Diaz, Ross Harmes'});
var myLibrary = new Library([bookOne, bookTwo]);
console.log('Library contains ' + myLibrary.length + ' books');
```

- The .length property allows you to get the number of models currently contained within the collection. Note that the .size() function returns the same number.

Manipulating Collections

ROHAN RAJORE

Adding Models

- Adding a new model to your collection can be easily done using the .add method.

```
var bookThree = new Book({name: 'Pro Node.js for Developers', author: 'Colin J. Ihrig'});
myLibrary.add(bookThree);
console.log('Library contains ' + myLibrary.length + ' books');
```

- The .add method will also accept an array of books.

```
var bookFour = new Book({name: 'Pro jQuery', author: 'Adam Freeman'});
var bookFive = new Book({name : 'Pro Javascript Performance', author: 'Tom Barker'});
myLibrary.add([bookFour, bookFive]);
console.log('Library has ' + myLibrary.length + ' books');
```

Adding Models

- If a model is already present in the collection when passed through to .add, it will be ignored. However, if {merge: true} is included in the call, the attributes will be merged into the duplicate model.

```
myLibrary.add(bookOne, {merge:true});
console.log('Library has ' + myLibrary.length + ' books');
```

- Note that an add event is fired when models are added to the collection.

Adding Models

- You can also use the .push() function to add a model to the end of the collection, providing either an array or a single model, as in the .add function.

```
myLibrary.push(bookFive);
console.log('Library has '+ myLibrary.length + 'books');
```

- To do the opposite of .push()and add the model to the beginning of the collection, use the .unshift function.

```
myLibrary.unshift(bookFive);
console.log('Library has '+ myLibrary.length + ' books');
```

Removing Models

- As you'd expect, a .remove function is available for removing a single modelor array of models.

```
myLibrary.remove(bookFive);
console.log('Library contains ' + myLibrary.length + ' books');
myLibrary.remove([bookThree, bookFour]);
console.log('Library contains ' + myLibrary.length + ' books');
```

Removing Models

- A remove event is fired when models are removed. An options object used in the listener can access the index of the element that has been removed.

```
var Library = Backbone.Collection.extend({model: Book,
initialize: function(){
this.on("remove", function(removedModel, models, options){
console.log('element removed at ' + options.index);
});
}
});
```

Removing Models

- The .pop() function removes and returns the last model in the collection.

```
var lastModel = myLibrary.pop();
```

- To remove the first model in the collection, use the .shift() function rather than .pop(). This will also return the model you are removing.

```
var firstModel = myLibrary.shift();
```

Resetting Collections

- The reset function exists to provide the ability to replace the set of models in a collection in a single call.

```
myLibrary.reset([bookOne]);
console.log('Library contains ' + myLibrary.length + ' books');
```

- You can empty the collection in one go by calling the reset method with no parameters.

```
myLibrary.reset();
console.log('Library contains ' + myLibrary.length + ' books');
```

- Using reset fires a single reset event rather than a sequence of remove and add events. This is useful for performance reasons because your application needs to react just once to a reset operation.

Smart Updating Collections

- The set function is described by the official Backbone documentation as a way to perform smart updates on a collection. By passing through an array of models, set abides by the following rules:
- If a model doesn't yet exist in the collection, it will be added. The rule will be ignored if {add: false} is provided.
- If a model is already in the collection, the attributes will be merged. The rule will be ignored if {merge: false} is provided.
- If there is a model in the collection that isn't in the array, it will be removed. The rule will be ignore if {remove: false} is provided.

Smart Updating Collections

- The following code snippet shows how the remove rule can be ignored:

```
myLibrary = new Library([bookOne, bookTwo]);
console.log('Library contains ' + myLibrary.length + ' books');
myLibrary.set([bookTwo], {remove: false});
console.log('Library contains ' + myLibrary.length + ' books');
```

- Without {remove:false}, the result of the second evaluation of myLibrary.length would have been 1.

Traversing Collections

ROHAN RAJORE

Retrieving Models

- Provided that you know the id of your models, you can retrieve a model from a collection using the .get function. Recall that until a model has been synchronized with a back-end service, the cid attribute is used instead.

```
var aBook = myLibrary.get('c5');
console.log('Retrieved book named ' + aBook.get('name'):
```

- If no model matches the id or cid that is used as a parameter, this function returns undefined.
- If you don’t want to use IDs, you can also use the .at function, which accepts the index at which the model is present in the collection. If the collection is not sorted, the index parameter will refer to the insertion order.

```
varanotherBook = myLibrary.at(1);
console.log('Retrieved book named ' + anotherBook.get('name'):
```

Iterating through Collections

- Although you can use a simple for loop to iterate through your collection as follows:

```
for(var i = 0; i < myLibrary.length; i++){
var model = myLibrary.at(i);
console.log('Book ' + i + ' is called ' + model.get('name'));
}
```

- there is a more elegant utility function provided by Underscore that helps iterate through collections, namely, the forEach function.

```
//using forEach
myLibrary.forEach(function(model){
console.log('Book is called ' + model.get('name'));
});
```

Utility Methods

ROHAN RAJORE

Sorting Collections

- Using the sortBy function, you can choose an attribute to use as a basis to sort your collection.

```
var sortedByName = myLibrary.sortBy(function (book) {
return book.get("name");
});
console.log("Sorted Version:");
sortedByName.forEach(function(model){
console.log('Book is called ' + model.get('name'));
});
```

- Note that sortBy returns a sorted array representation of the models in the collectionand doesn’t actually change the order of the models within the collection.

Sorting Collections

- By utilizing comparators, you can impose a sorted order that will always be used for your collection. It’s probably most useful to define your comparator function during collection definition. The following example illustrates how to create a comparator to order books by name:

```
var Library = Backbone.Collection.extend({model: Book,
initialize: function(){
//initialize function content..
},
comparator: function(a, b) {
return a.get('name') < b.get('name') ? -1 : 1;
}
});
```

Sorting Collections

- The collection will now always be sorted by name. However, if you change an attribute value in one model, the collection will not rearrange the ordering. Instead, the order can be applied by invoking the sort function for the collection.

```
myLibrary.at(0).set('name', 'Z');
myLibrary.forEach(function(model){
console.log('Book is called ' + model.get('name'));
});
//force sort
myLibrary.sort();
myLibrary.forEach(function(model){
console.log('Book is called ' + model.get('name'));
});
```

- Note that the sorting order will be reapplied when a new model is added to the collection, unless the {sort: false} option is passed through when adding.

Shuffle

- If you need to get a randomized version of the models in your collection, the shuffle() function will return an array of the models that have had a shuffling algorithm applied.

```
varshuffled = myLibrary.shuffle();
```

- Iterating through the collection now will present the books in a different order than before.

```
myLibrary.forEach(function(model){
console.log('Book is called ' + model.get('name'));
});
```

Getting a list of Attributes

- Sometimes you may want to get a list of all the instances of a particular attribute in your collection. This can be done easily using the .pluck() function.

```
console.log('List of authors in collection:');
var authors = myLibrary.pluck('author');
authors.forEach(function(authorName){
console.log(authorName)
});
```

Searching

- A number of useful search mechanisms are available, based on passing through a set of key-value pairs to search functions.
- For example, to get an array of model objects that match a certain criteria, use the .where function.

```
var results = myLibrary.where({author:'James Sugrue'});
console.log('Found: ' + results.length + ' matches');
```

- The findWhere function can be used to find the first model that matches the query, rather than returning an array of matches.

```
var result = myLibrary.findWhere({author:'James Sugrue'});
console.log('Result: ' + result.get('author'));
```

Searching

- You can also group model objects by common attribute values. For example, if each of the books had a Boolean indicating whether they were published, we could group them as follows:

```
var byPublished = myLibrary.groupBy('published');
myLibrary.forEach(function(model){
console.log('Book is called ' + model.get('name'));
});
```

Exchanging Data with the Server

ROHAN RAJORE

Setup

- The first thing you need to do is tell the collection where it needs to point to for its data persistence, using the url attribute. We’ll be pointing to our Node.js server, running a /books/ API endpoint on port 8080.

```
var Library = Backbone.Collection.extend({model: Book,
url: 'http://localhost:8080/books/',
initialize: function(){
....
```

Retrieving Data from Server

- Just as with Backbone.Model, collections have a .fetch method that is used to retrieve the collection data from the server.

```
var myOnlineLibrary = new Library();
myOnlineLibrary.fetch({
success: function(e){
console.log('Got data');
},
error: function(e){
console.log('Something went wrong');
}
});
```

Retrieving Data from Server

- As with all functions that deal with online operations, you can provide success and error callbacks to detect whether the operation completed correctly.
- Similar to Backbone.Model, a parse method can be defined at the collection level, which allows you to customize the data returned from the server. The response parameter contains an array of all model objects that are retrieved after the execution of .fetch.

```
var Library = Backbone.Collection.extend({model: Book,
......
parse: function(response, xhr) {
//customisations here
return response;
},
});
```

Saving Data to the Server

- While retrieving data is done in a batch, saving models to the server is still done onan individual basis, as in the Backbone.Model sections of this chapter.

```
//add a model to the collection and save to server
myOnlineLibrary.add(bookOne);
bookOne.save({
success: function(e){
console.log('Book saved to server');
},
error: function(e){
console.log('Error saving book);
}
});
```

Saving Data to the Server

- However, if you use the Backbone.Collection.create function rather than .add, the model is both added to the collection and persisted to the server.

```
//add to the collection and save all at once
myOnlineLibrary.create(bookTwo);
```

Deleting data from the Server

- Removing models from the server is done at the individual model level using the .destroy method, as described in the previous section.

```
bookOne.destroy({
success: function(e){
console.log('Book deleted')
},
error: function(e){
console.log('Error deleting book');
}
});
```

Collection Events

- **add**: Detects when a model has been added to the collection
- **remove**: Detects when a model has been removed from a collection
- **reset**: Detects that the collection has been reset
- **sort**: Detects that the collection is being sorted
- **change**: Detects that a model within the collection is being changed.
- **change:<attribute name>**: Detects that <attribute name>, a model within the collection, is being changed