Models

ROHAN RAJORE

Models and Collections

- There are two main parts to the model layer of any Backbone.js application:
  - the representation of the data (Backbone.Model)
  - And the grouping of your data into sets (Backbone.Collections).
- An understanding of how each of these parts works and interacts will give you the ability to create a well-structured data layer for your Backbone applications.

Backbone Models

- Every application begins with the definition of a model. It's the data that you need to manipulate and represent in the app.
- The model is a layer in which the data resides and will likely contain many objects.
- As a rule of thumb, you should break your problem domain into individual elements and describe each of these as a model object.
- To create a model object in Backbone, you will need to use the Backbone.Model. As you'll see for all Backbone objects that you create, you'll use an .extend method to create your own version of the class, as in the following code:

```
MyModel = Backbone.Model.extend({
//object properties would go here
});
```

Constructors

- When creating new model objects, you may want some behavior to be executed on construction, such assetting up different handlers or setting some initial state in your mode. To achieve this in Backbone, just define an initialize function when creating your model object.
- To see this initialize function in action, let's create a simple Book object where a line is written to the console on object creation.

```
Book = Backbone.Model.extend({
initialize: function(){
console.log('a new book');
}
});
```

- This initialize function gets called as soon as we create a new instance of the Model object. In this case, on creation of a new Book object, we'll see the line "a new book" on the console.

```
varmyBook = new Book();
```

Constructors

- Another common requirement for models is to have some default attributes available. You may want to do this so that optional parameters, not passed through on object creation, have some definition. In Backbone, this is done using the defaults object literal.

```
Book = Backbone.Model.extend({
initialize: function(){
console.log('a new book');
},
defaults: {
name: 'Book Title',
author: 'No One'
}
});
```

- Now when the object is created, these default values are provided for each instance.

Model Attributes

- This section coversall the attribute-related operations that are available to you when using Backbone.Model.
- Setting Attribute Values on Object Creation:

```
varthisBook = new Book({name : 'Beginning Backbone',
author: 'James Sugrue'});
```

Getting Attribute Values

- The attributes in any model object can be easily retrieved by using the .get function and passing through the name of the attribute that you want to access.

```
console.log(myBook.get('name'));
```

- You can also use the .attributes property to get a JSON object that represents all of the model data.

```
console.log(thisBook.attributes);
```

Changing Attribute Values

- Changing attribute values outside of the constructor is done in a similar way, using the structure of a function call in the format .set('<variable name>', <value>). For example, to change the book name, use this:

```
thisBook.set('name', 'Beginning Backbone.js');
```

- You can add new attributes to your object in the same manner as shown previously.

```
thisBook.set('year', 2013);//creates a new attribute called year
```

Deleting Attributes

- You may find that you need to delete attributes you have no use for from your model. In the following example, a new attribute is added and subsequently deleted. The second console.log statement results in undefined being returned as the attribute value.

```
thisBook.set('year', 2013);
console.log('Book year ' + thisBook.get('year'));
thisBook.unset('year');
console.log('Book year ' + thisBook.get('year'));
```

- You can also delete all attributes from your model using the clear function.

```
varnewBook = new Book({name: 'Another Book', author: 'You'});
newBook.clear();//remove all attributes
console.log('Has an attribute called name : ' +
newBook.has('name'));//results in false
```

Checking for the presence of an Attribute

- Backbone provides a neater way to check for the presence of an attribute in a model, using the .has function, which returns a Boolean.

```
//check for the existence of an attribute
varhasYear = thisBook.has('year'); //results in false
varhasName = thisBook.has('name'); //results in true
console.log('Has an attribute called year : ' + hasYear);
console.log('Has an attribute called name : ' + hasName);
```

Cloning Models

- It’s common that you might want to make a complete copy of your Backbone model, keeping all the same attributes. Rather than needing to worry about the details of how to create a deep copy, you can simply use the .clone() method to create a cloned model instance.

```
varclonedBook = thisBook.clone();
```

Adding functions to your model

- So far our model has been all about the attributes, but you can also add your own functions to deal with repetitive tasks. The following example illustrates how to include a printDetails function in place of all the console.log statements used so far:

```
Book = Backbone.Model.extend({
    initialize: function(){
        console.log('a new book');
    },
    defaults: {
        name: 'Book Title',
        author: 'No One'
    },
    printDetails: function(){
         console.log(this.get('name') + ' by ' + this.get('author'));
    }
});
```

Adding functions to your model

- This function is now available to all instances of the Book object.

```
//use the printDetails function
thisBook.printDetails();
```

Model Events: Listening for change Events

- With Backbone change handlers, the easiest type of change handler listens for changes across the entire model. Event handlers are added to Backbone objects using the .on() function, which accepts the type of handler as a string andaccepts a reference to a function that will be run when the change happens.
- The best time to create this listener is in the model initialize function. By altering the code you have so far, you will see that any time a set is called, this function is invoked.

```
Book = Backbone.Model.extend({
    initialize: function(){
        this.on("change", function(){
            console.log('Model Changed');
        });
    }
});
```

Model Events: Listening for change Events

- You can listen for changes in specific attributes by using the format change:<attribute name> rather than change. The following addition creates another handler that deals only with changes to the name attribute:

```
initialize: function(){
    this.on("change", function(){
        console.log('Model Changed');
    });
    this.on("change:name", function(){
        console.log('The name attribute has changed');
    });
}
```

Silent Updates

- We noted earlier that the .set function allowed an optional parameter for silent updates. If this is used, then the change handler won't be invoked.

```
//set the variable (expect change handler)
thisBook.set('name','Different Book'); //change handler invoked
thisBook.set(,'name', 'Different Book', {silent:true});//no change handler invoked
```

Figuring Out What Has Changed

- Backbone includes a number of properties thatkeep track of what is changed in your model. If you are using a global change handler, this can be a really useful way to see what's going on.
- You can check whether an individual attribute has been altered using hasChanged('<attribute name'>).

```
this.on("change", function(){
    console.log('Model Changes Detected:');
    if(this.hasChanged('name')){
        console.log('The name has changed');
    }
    if(this.hasChanged('year')){
        console.log('The year has changed')
    }
});
```

All attributes that have changed

- You can get a set of all the attributes that have changed using the .changed property.

```
Book = Backbone.Model.extend({
    initialize: function(){
        this.on("change", function(){
        console.log('Changed attributes: ' + JSON.stringify(this.changed));
    });
}
```

Model Validation

- Backbone provides a validation mechanism for model data, meaning that you can have all the logic that determines whether the state of the model is correct or not within the model, rather than in some external JavaScript or form-processing code.
- If you provide a validation method, it will be run every time the .save function is invoked and during every set/unset operation when {validate:true} is provided as an optional parameter.

Model Validation

- Let's imagine our Book model insists that a name exists and that the year is after 2000. A validation method for these rules would look as follows:

```
Book = Backbone.Model.extend({
    validate: function(attrs){
        if(attrs.year< 2000){
            return 'Year must be after 2000';
        }
        if(!attrs.name){
            return 'A name must be provided';
        }
    }
});
```

Model Validation Errors

- If you break any of these rules when manipulating the model, the operations will fail to change the attribute values.

```
//try setting the year to pre 2000
thisBook.set('year', 1999, {validate: true});
console.log('Check year change: ' + thisBook.get('year'));
//try removing the name from the model
thisBook.unset('name', {validate: true});
console.log('Check if name was removed ' + thisBook.get('name'));
```

- Without the validation flag, the validation function will not be executed on set. However, you can check whether the model is in a valid state at any time with the isValid() function.

```
//check if model is valid
console.log('Is model valid: ' + thisBook.isValid());
```

Model Validation Errors

- When a validation error has been detected, an event is fired. By adding an “invalid” event handler, you can provide feedback on the validation error. As with all event handlers, this should be added to your initialize function.

```
Book = Backbone.Model.extend({
    initialize: function(){
        this.on("invalid", function(model, error){
        console.log("**Validation Error : " + error + "**");
});
```

Identifiers

- Backbone models have three attributes that deal with uniquely identifying them during data exchange with the server: id, cid, and idattribute.
- The id attribute is a unique string or integer value, just likea primary key in a relational database. This id attribute is useful when retrieving the model from a collection, and it is also used to form part of the URL for the model.
- The cid attribute is generated automatically by Backbone when the model is first created; it can be used to serve as a unique identifier when the model has not yet been saved to the server and does not have its real ID available.
- Sometimes the model you are retrieving from the backend will use a different unique key. For example, the server might use an ISBN as the unique identifier for a book, or a user IDfield might be the identifier used for a User model when saved. The idAttribute attribute allows you to provide a mapping between that key to the ID in your model, meaning that the server will use that attribute to populate the ID.

Saving Models

- The save function invokes the operation to save the model to the server.
- In cases where the id attribute has not been set and save is called, the model will invoke acreate operation (HTTP POST) on the back-end REST service, while an update (HTTP PUT) operation will be used when the ID has been specified. This is a simple way of ensuring that a single save function can be used regardless of whether your model has been newly created or has been edited since last retrieved from the server.
- The save function can be called with no parameters or can take the set of attributes you want to persist to the server, along with an options hash that contains handlers for both success and error cases.

Saving Models

```
thisBook.save(thisBook.attributes,
 {
      success: function(model, response, options){
         console.log('Model saved');
         console.log('Id: ' +thisBook.get('id'));
      },
      error: function(model, xhr, options){
         console.log('Failed to save model');
      }
});
```

- Once the call is complete and has returned, the appropriate callback will be invoked, either success or error. Calls are made asynchronously, so any lines of code after the save method won't wait for the save to be completed first.
- Note that the validation function will be called during the execution of save(). If the validation fails, the model will not be sent to the server.

Retrieving Models

- If you want to reset the state of your model object to the same as it is on the server side, you can invoke the fetch() function. Again, this function accepts an options hash that includes success and error callbacks.

```
thisBook.fetch({
        success: function(model, response, options){
           console.log('Fetch success');
        },
        error: function(model, response, options){
           console.log('Fetch error');
        }
});
```

- If the execution of the fetch function detects that there is a difference in the models between the server and client sides, a change event will be triggered. This can be useful when you want to ensure that the application is in sync with the back-end service or when you need to populate your model objects on application start-up.

Deleting Models

- The final server operation that you may want to carry out is a delete operation to remove the model from the backend.

```
thisBook.destroy({
  success: function(model, response, options){
     console.log('Destroy success');
  },
  error: function(model, response, options){
     console.log('Destroy error');
  },
  wait: true
});
```

- If the model is new and doesn't yet exist on the server, the destroy operation will fail. Adding a wait:true option will ensure that the model is successfully removed from the server before it is removed from any Backbone.Collection that contains it on the client side.

Extending Model

- As you'll recall, when creating new models, you need to use Backbone.Model.extend in the creation.

```
Book = Backbone.Model.extend({
```

- You can use the same format to extend this further. In the followingexample, let's create an EBook model, extending the original Book model with an additional method:

```
EBook = Book.extend({
getWebLink: function(){
return 'http://www.apress.com/'+this.get('name');
}
});
var ebook = new EBook({name: "Beginning Backbone", author: "James Sugrue"});
console.log(ebook.getWebLink());
```

Extending Model

- To call a function in the parent class, you need to call it explicitly using the prototype and passing through this so that the current model is used as the context.

```
EBook = Book.extend({
printDetails: function(){
console.log('An ebook');
Book.prototype.printDetails.call(this);
}
});
```

- The previous code results in the printDetails function from the Book model being executed after an additional print statement defined in the EBook model is displayed. If there were any parameters involved in this function, they would be passed along after the this reference.