

# SINGLE PAGE APPLICATION: BACKBONE JS

## ADVANCED JAVASCRIPT: CA #2

Jonathan Dempsey  
Advanced JavaScript  
April 6, 2016

## How to Start

Open two command prompts. In command prompt one: navigate to the folder which holds the MongoDB file and enter `mongodb\server\3.2\bin\mongod.exe` and leave it running. In command prompt two: navigate to the `'gameWishList'` folder and enter `node server` and leave it running.

In `'gameWishList'` navigate to `'public'` -> `'gameWishList'` -> `'index'` to open the HTML page. With the two command prompts running it should function normally.

***\*\*Important\*\**** Before Easter I had it working normally but now an error occurs saying *"Access-Control-Allow-Origin' header is present on the requested resource. Origin 'null' is therefore not allowed access."* To fix this I had to download an extension to enable cross-origin resource sharing called Allow-Control-Allow-Origin: \*. The application doesn't seem to work without the extension running.

## Introduction

A single page HTML/CSS/JavaScript web application using the JavaScript MVC (Model, View Controller) framework, Backbone JS. The application dynamically views, adds, deletes and updates the page content, interacts with a back-end web server and handles events within the page. The application is made using code from a tutorial by Christophe Coenraets at <http://coenraets.org/blog/2012/10/creating-a-rest-api-using-node-js-express-and-mongodb/>

To begin, a RESTful API is created using Node.js, Express and MongoDB.

- Node.js is a JavaScript run time which uses an event-driven and non-blocking I/O model, which is useful building scalable network applications. Once node is downloaded off nodejs.org, it's ready to be implemented.
- Express is a Node.js web application framework which will allow our web application to dynamically render the HTML page based on arguments passed to templates. Express can be installed globally by entering `$ npm install express --save` into the command line.
- MongoDB is a simple to use database that uses a JSON data model which maps to applications, and has dynamic schemas that allow quick iterations. To implement mongo, a data directory needs to be created to store all the data. To create the folder, enter `md \data\db` into the command prompt. To start mongo, execute mongod.exe by entering `mongodb\server\3.2\bin\mongod.exe` into the command prompt.

## Execution

To execute the application, two command prompts are required. The first command prompt will run MongoDB. This can be done by navigating through the mongodb folder to execute mongod.exe. As shown in Figure 1, mongodb file was located on the desktop.

```
C:\Users\N00112462\Desktop>mongodb\server\3.2\bin\mongod.exe
```

Figure 1: Execute MongoDB

The second prompt is used to run server.js, which is a JavaScript file that implements all the routes required by the API and will be detailed later in the report. To execute, navigate to the parent folder directory and initiate server.js, as seen in Figure 2.

```
C:\Users\N00112462\Desktop\gameStore>node server
```

Figure 2: Execute Server.js

## Server.js & Package.json

Server.js is used to implement the routes used in the application. Two variables are created using 'require()' (Fig 3). Nodejs uses require() to organise programs and libraries into self-contained directories, which provides a single entry point to that library/program.

```
var express = require('express'),  
    game = require('./routes/games');
```

Figure 3: require() Variables

The first variable requires 'express', which access a file called package.json. Package.json is a small file that defines dependencies within the application, in this case being Express and MongoDB (Fig 4).

```
{  
  "name": "game-list",  
  "description": "Game List Application",  
  "version": "0.0.1",  
  "private": true,  
  "dependencies": {  
    "express": "3.x",  
    "mongodb": "^2.1.4"  
  }  
}
```

Figure 4: Package.json

The second variable creates a route to the games.js, which acts as the database model and provides the data access logic for each route.

Server.js then creates a variable called 'app', which initiates Express. App.use accesses express:

- Logger: HTTP request logger middleware for node.js.
- Static: Serves files from within a given directory (in this case /public). If a file isn't found, instead of a 404 response, it calls *next()* allowing for stacking and fall-backs.
- bodyParser: Exposes factories to create middlewares, allowing middlewares to populate the *req.body* property
- Get/post/put/delete: access games.js and pulls the findAll and findById, addGame, updateGame and deleteGame methods respectively.
- Listen: opens up port 3000 which can be accessed through localhost

## Game.js

Four variables are created at the beginning of the file. These variables initiate Mongo as the applications server and BSON which is a format mainly used as data storage and a network transfer format in MongoDB. The server is then connected to localhost and a new database is created called gamedb which connects to localhost. When the database is opened, it's collection of data is checked. If the collection is empty, the database is populated with sample data (Fig 5).

```
1  var mongo = require('mongodb');
2  var bson = require('bson');
3
4  var Server = mongo.Server,
5      Db = mongo.Db,
6      BSON = bson.BSONPure;
7
8  var server = new Server('localhost', 27017, {auto_reconnect: true});
9  db = new Db('gamedb', server);
10
11 db.open(function(err, db) {
12   if(!err) {
13     console.log("Connected to 'gamedb' database");
14     db.collection('games', {strict:true}, function(err, collection) {
15       if (err) {
16         console.log("The 'games' collection doesn't exist. Creating it with sample data...");
17         populateDB();
18       }
19     });
20   }
21 });
```

Figure 5: Server & Database Setup

The view, add, update and delete functions are initiated. They all follow the same general layout so the update function will be used as an example.

The function is put into an object, in this case `exports.updateGame`. Export allows objects to become available to use in other files (Fig 6).

The function is then given two parameters, req and res. Req (request) is an object containing information about the HTTP request. Res (response) is a response to req which sends back the desired HTTP response (Fig 6).

```
exports.updateGame = function(req, res) {
```

Figure 6: Initiate Function

The functions then create variables, id and game. Var id sends a request for the parameter of the object in the databases' id. Var game requests the information from the body of the web page. The game id and the updated data in JSON format is then printed into the console (Fig 7).

```
var id = req.params.id;
var game = req.body;
console.log('Updating game: ' + id);
console.log(JSON.stringify(game));
```

Figure 7: Requests

The games database is then opened. The collection uses `.update` followed by the objects id being appended into a BSON object, which allows it to be passed into MongoDB database. If there's an error the response will send an error message and if there's no error the response will send var game. For other functions, collection.update is replaced with `.findOne`, `.find`, `.insert` or `.remove` (Fig 8).

```
db.collection('games', function(err, collection) {
  collection.update({'_id':new BSON.ObjectId(id)}, game, {safe:true}, function(err, result) {
    if (err) {
      console.log('Error updating game: ' + err);
      res.send({'error':'An error has occurred'});
    } else {
      console.log('' + result + ' document(s) updated');
      res.send(game);
    }
  });
});
```

Figure 8: Commit Function & Error Checking

Finally, a variable is created which contains two items in an array which are to be added to the database as sample data if the database is empty. In this case, the sample data contain the game 'Bioshock Infinite' and 'XCOM 2' (Fig 9).

```

var populateDB = function() {

  var games = [
    {
      name: "Bioshock Infinite",
      releaseDate: "March 26, 2013",
      publisher: "Irrational Games, 2K Marin, Aspyr Media, Inc., 2K Australia",
      console: "PlayStation 3, Xbox 360, Microsoft Windows, Linux, Mac OS",
      price: "49.99",
      description: "...",
      picture: "Bioshock Infinite.jpg"
    },
    {
      name: "XCOM 2",
      releaseDate: "February 5, 2016",
      publisher: "Firaxis Games",
      console: "Microsoft Windows, Linux, Mac OS",
      price: "49.99",
      description: "...",
      picture: "XCOM2.jpg"
    }
  ];

  db.collection('games', function(err, collection) {
    collection.insert(games, {safe:true}, function(err, result) {});
  });
};

```

Figure 9: Sample Data

## Main.js & index.html

Index.html is primarily used as a skeleton for main.js. A series of templates are added in the HTML which get populated by the data in the database through backbone. Other than templates, backbone uses `<%= value %>` to access specific values from objects in the database. Figure 10 shows the list that gets populated on the left which gets the objects Id and displays the objects name. Figure 11 shows the selected objects list of values as well as providing buttons for updating and deleting.

```

<script type="text/template" id="tpl-game-list-item">
  <a href="#games/<%= _id %>"><%= name %></a>
</script>

```

Figure 10: Game List Template

```

<script type="text/template" id="tpl-game-details">

  <div class="form-left-col">
    <label>Id:</label>
    <input type="text" id="gameId" name="_id" value="<%= _id %>" disabled />
    <label>Name:</label>
    <input type="text" id="name" name="name" value="<%= name %>" required/>
    <label>Publisher:</label>
    <input type="text" id="publisher" name="publisher" value="<%= publisher %>"/>
    <label>Console:</label>
    <input type="text" id="console" name="console" value="<%= console %>"/>
    <label>Price:</label>
    <input type="text" id="price" name="price" value="<%= price %>"/>
    <label>Release Date:</label>
    <input type="text" id="releaseDate" name="releaseDate" value="<%= releaseDate %>"/>
    <button class="save">Save</button>
    <button class="delete">Delete</button>
  </div>

  <div class="form-right-col">
    
    <label>Notes:</label>
    <textarea id="description" name="description"><%= description %></textarea>
  </div>

</script>

```

Figure 11: Game Details Template

Main.js is used to establish the model, view and routes to be used in the HTML.

Backbone first establishes the model of the game object which will be used to send and receive game objects to the database, it then gets the collection of objects using the same model from port 3000 (Fig 12).

```

// Models
window.Game = Backbone.Model.extend({
  idAttribute: "_id",
  urlRoot: "http://localhost:3000/games",
  defaults: {
    "_id": null,
    "name": "",
    "publisher": "",
    "console": "Xbox",
    "price": "49.99",
    "releaseDate": "",
    "description": "",
    "picture": ""
  }
});

window.GameCollection = Backbone.Collection.extend({
  model: Game,
  url: "http://localhost:3000/games"
});

```

Figure 12: Games Model



Backbone then creates a series of templates which will send and receive data from the database to be displayed in the template skeletons the HTML code established.

First a parent view is initialised and rendered. Once initialised, the rendering is done to ensure that any elements the child views depend on already exist before they are assigned. This allows child events that occur to be set correctly which can then be re-rendered without having to worry about re-delegating anything. An element ( *el* ) is passed into the parent which will determine what the child view is allowed to input into the parent (*Fig 13*).

```
window.GameListView = Backbone.View.extend({
  tagName: 'ul',

  initialize: function () {
    this.model.bind("reset", this.render, this);
    var self = this;
    this.model.bind("add", function (game) {
      $(self.el).append(new GameListItemView({model: game}).render().el);
    });
  },

  render: function (eventName) {
    this.each(this.model.models, function (game) {
      $(this.el).append(new GameListItemView({model: game}).render().el);
    }, this);
    return this;
  }
});
```

*Figure 13: Parent Initialise and Render*

The child views are then initialised and rendered. The binds used in the initialiser means that render doesn't need to be called manually, as the model block will re-render itself if it detects any change and won't affect other views. The child also contains a close, which will unbind and remove an element from the model (*Fig 14*).



```

window.GameListItemView = Backbone.View.extend({

  tagName: "li",

  template: _.template($('#tpl-game-list-item').html()),

  initialize: function () {
    this.model.bind("change", this.render, this);
    this.model.bind("destroy", this.close, this);
  },

  render: function (eventName) {
    $(this.el).html(this.template(this.model.toJSON()));
    return this;
  },

  close: function () {
    $(this.el).unbind();
    $(this.el).remove();
  }

});

```

Figure 14: Child Initialised, Rendered & Closed

The game detail view is with the same initialise and render functions as before, but has the addition of three events: change input, click .save and click .delete.

The change function allows the model to be changed on the go through console commands (Fig 15).

```

change: function (event) {
  var target = event.target;
  console.log('changing ' + target.id + ' from: ' + target.defaultValue + ' to: ' + target.value);
}

```

Figure 15: Game Detail Change

The save function allows the model object to be updated and allows new ones to be added to the database. The function reads the values entered into each of the text boxes from the HTML, if the object already exists it overwrites the old values in the database to the new values, and if it's a new object, it simply adds an additional object into the database to be displayed (Fig 16).

```

saveGame:function () {
    this.model.set({
        name:$('#name').val(),
        publisher:$('#publisher').val(),
        console:$('#console').val(),
        price:$('#price').val(),
        releaseDate:$('#releaseDate').val(),
        description:$('#description').val()
    });
    if (this.model.isNew()) {
        var self = this;
        app.gameList.create(this.model, {
            success:function () {
                app.navigate('games/' + self.model.id, false);
            }
        });
    } else {
        this.model.save();
    }

    return false;
},

```

Figure 16: Save Function

The delete function will remove objects from the database. When on the specific object, when delete button is clicked, `.destroy` is used to remove the selected model object. If successful, `window.history.back()` is used to get back a page (same as clicking the back button), and if not successful, nothing happens (Fig 17).

```

deleteGame:function () {
    this.model.destroy({
        success:function () {
            alert('Game deleted successfully');
            window.history.back();
        }
    });
    return false;
},

close:function () {
    $(this.el).unbind();
    $(this.el).empty();
}

```

Figure 17: Delete Function

A new view is created for the new function and follows the same general structure as the game detail view. It gets initialised and rendered and gets an event *click . new* added. Clicking navigates the application to “games/new” which is a route that gets defined later in the code (Fig 18).

```
window.HeaderView = Backbone.View.extend({
  template:_.template($('#tpl-header').html()),
  initialize:function () {
    this.render();
  },
  render:function (eventName) {
    $(this.el).html(this.template());
    return this;
  },
  events:{
    "click .new":"newGame"
  },
  newGame:function (event) {
    app.navigate("games/new", true);
    return false;
  }
});
```

Figure 18: New Function

The Routers in main.js are used to navigate render data from the database into HTML id's. Three routers are used:

The first renders data from 'gamelist' into the 'sidebar' element in the html. Backbone first creates a new *GameCollection()*; and then uses *fetch()* to render the fetched data from the database into the HTML element 'sidebar' (Fig 19). Figure 20 shows how it appears within the HTML.

```
list:function () {
  this.gameList = new GameCollection();
  var self = this;
  this.gameList.fetch({
    reset: true,
    success:function () {
      self.gameListView = new GameListView({model:self.gameList});
      $('#sidebar').html(self.gameListView.render().el);
      if (self.requestedId) self.gameDetails(self.requestedId);
    }
  });
},
```

Figure 19: List Function

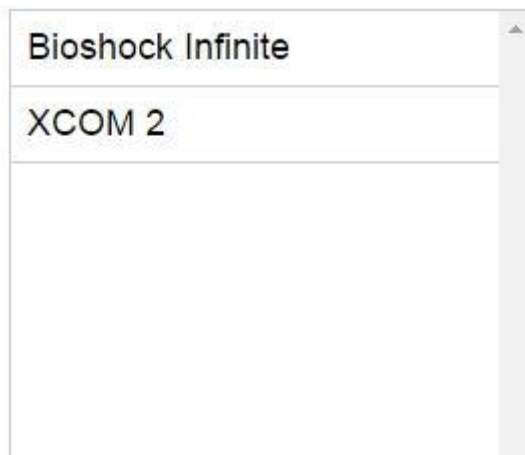


Figure 20: Sidebar in with Rendered Data

The next function loads the details about the selected game into the 'content' element in the HTML. The function passes the id parameter which reads the id of the selected object in the sidebar, sends a get request to retrieve additional data (such as price, publisher and image) relevant to that id, and then renders that data into the 'content' element (Figure 21). Figure 22 shows how it appears within the HTML.

```
gameDetails:function (id) {  
  if (this.gameList) {  
    this.game = this.gameList.get(id);  
    if (this.gameView) this.gameView.close();  
    this.gameView = new GameView({model:this.game});  
    $('#content').html(this.gameView.render().el);  
  } else {  
    this.requestedId = id;  
    this.list();  
  }  
},
```

Figure 21: Game Details Function


Id:	56fabdd8b611ddf022c79941	
Name:	Bioshock Infinite	
Publisher:	Irrational Games	
Console:	Xbox 360	
Price:	49.99	
Release Date:	March 26, 2013	
<input type="button" value="Save"/> <input type="button" value="Delete"/>		
Notes:		BioShock Infinite is a first-person shooter video game developed by Irrational Games and published by 2K Games

Figure 22: Content with Rendered Data

The final function is used to add new objects. It closes the game details view from the 'content' element and renders in the default model which was defined at the beginning of the file into the content element instead (Fig 23). Figure 24 shows how it appears within the HTML.

```
newGame:function () {
  if (app.gameView) app.gameView.close();
  app.gameView = new GameView({model:new Game()});
  $('#content').html(app.gameView.render().el);
}
```

Figure 23: New Game Function

Id: 

Name:

Publisher:

Console:

Price:

Release Date:

Notes:  

...

Figure 24: Default Model Rendered in HTML