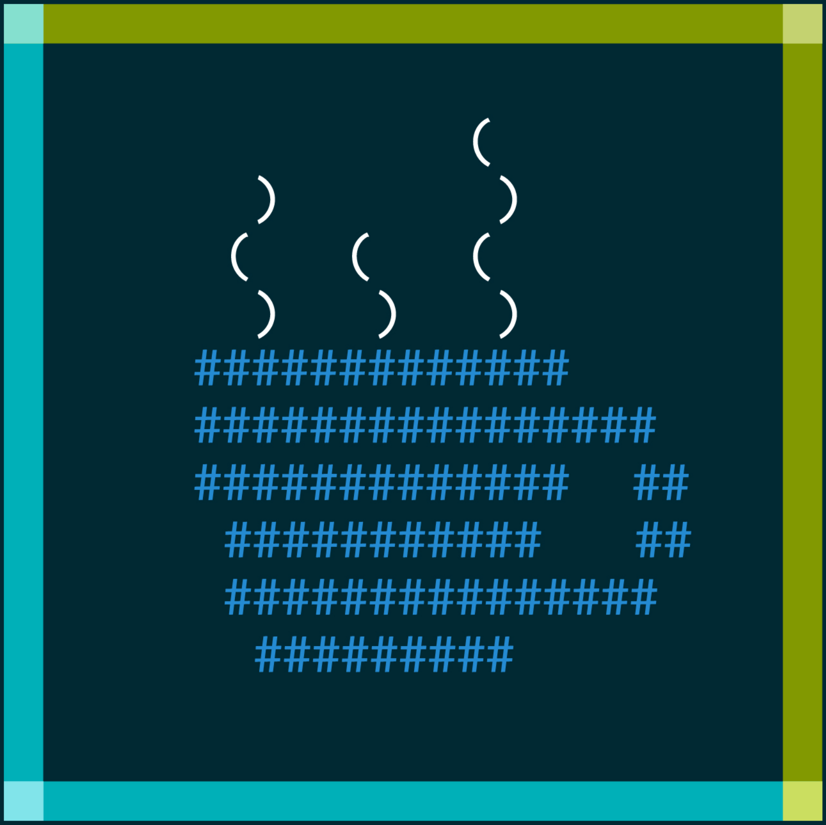
Start learning Backbone.js, Node.js and MongoDB

[AZAT MARDAN]



RAPID PROTOTYPING WITH JS

AGILE JAVASCRIPT DEVELOPMENT

Rapid Prototyping with JS

Agile JavaScript Development

Azat Mardan

This book is for sale at http://leanpub.com/rapid-prototyping-with-js

This version was published on 2015-05-05

Tweet This Book!

Please help Azat Mardan by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I've downloaded Rapid Prototyping with JS — book on JavaScript and Node.js by @azat_co #RPJS @RPJSbook

The suggested hashtag for this book is #RPJS.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#RPJS

Contents

1. Intro to Backbone.js

Summary: demonstration of how to build Backbone.js application from scratch and use views, collections, subviews, models, event binding, AMD, Require.js on the example of the apple database application.

“Code is not an asset. It’s a liability. The more you write, the more you’ll have to maintain later.”
— Unknown

1.1 Setting up Backbone.js App from Scratch

We’re going to build a typical starter “Hello World” application using Backbone.js and Mode-View-Controller (MVC) architecture. I know it might sound like overkill in the beginning, but as we go along we’ll add more and more complexity, including Models, Subviews and Collections.

A full source code for the “Hello World” app is available at GitHub under github.com/azat-co/rpjs/backbone/hello-world[1].

1.1.1 Dependencies

Download the following libraries:

- jQuery 1.9 development source file[2]
- Underscore.js development source file[3]
- Backbone.js development source file[4]

And include these frameworks in the **index.html** file like this:

[1]https://github.com/azat-co/rpjs/tree/master/backbone/hello-world
[2]http://code.jquery.com/jquery-1.9.0.js
[3]http://underscorejs.org/underscore.js
[4]http://backbonejs.org/backbone.js

```html
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
    //TODO write some awesome JS code!
  </script>

</head>
<body>
</body>
</html>
```

Note

We can also put `<script>` tags right after the `</body>` tag in the end of the file. This will change the order in which scripts and the rest of HTML are loaded, and impact performance in large files.

Let's define a simple Backbone.js Router inside of a `<script>` tag:

```
  ...
  var router = Backbone.Router.extend({
  });
  ...
```

Note

For now, to Keep It Simple Stupid (KISS), we'll be putting all of our JavaScript code right into the **index.html** file. This is not a good idea for a real development or production code. We'll refactor it later.

Then set up a special `routes` property inside of an **extend** call:

```
  var router = Backbone.Router.extend({
    routes: {
    }
  });
```

The Backbone.js **routes** property needs to be in the following format: `'path/:param':'action'` which will result in the `filename#path/param` URL triggering a function named **action** (defined in the Router object). For now, we'll add a single **home** route:

```
var router = Backbone.Router.extend({
  routes: {
    '': 'home'
  }
});
```

This is good, but now we need to add a **home** function:

```
var router = Backbone.Router.extend({
  routes: {
    '': 'home'
  },
  home: function(){
    //TODO render html
  }
});
```

We'll come back to the **home** function later to add more logic for creating and rendering of a View. Right now we should define our **homeView**:

```
var homeView = Backbone.View.extend({
});
```

It looks familiar, right? Backbone.js uses similar syntax for all of its components: the **extend** function and a JSON object as a parameter to it.

There are a multiple ways to proceed from now on, but the best practice is to use the **el** and ***template*** properties, which are magical, i.e., special in Backbone.js:

```
var homeView = Backbone.View.extend({
  el: 'body',
  template: _.template('Hello World')
});
```

The property **el** is just a string that holds the jQuery selector (you can use class name with '.' and id name with '#'). The template property has been assigned an Underscore.js function **template** with just a plain text 'Hello World'.

To render our **homeView** we use `this.$el` which is a compiled jQuery object referencing element in an **el** property, and the jQuery `.html()` function to replace HTML with `this.template()` value. Here is what the full code for our Backbone.js View looks like:

```
  var homeView = Backbone.View.extend({
    el: 'body',
    template: _.template('Hello World'),
    render: function(){
      this.$el.html(this.template({}));
    }
  });
```

Now, if we go back to the **router** we can add these two lines to the **home** function:

```
  var router = Backbone.Router.extend({
    routes: {
      '': 'home'
    },
    initialize: function(){

    },
    home: function(){
      this.homeView = new homeView;
      this.homeView.render();
    }
  });
```

The first line will create the *homeView* object and assign it to the *homeView* property of the router. The second line will call the `render()` method in the *homeView* object, triggering the 'Hello World' output.

Finally, to start a Backbone app, we call `new Router` inside of a document-ready wrapper to make sure that the file's DOM is fully loaded:

```
var app;
$(document).ready(function(){
  app = new router;
  Backbone.history.start();
})
```

Here is the full code of the **index.html** file:

```html
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
    var app;
    var router = Backbone.Router.extend({
      routes: {
        '': 'home'
      },
      initialize: function(){
       //some code to execute
       //when the object is instantiated
      },
      home: function(){
        this.homeView = new homeView;
        this.homeView.render();
      }
    });

    var homeView = Backbone.View.extend({
      el: 'body',
      template: _.template('Hello World'),
      render: function(){
        this.$el.html(this.template({}));
      }
    });

    $(document).ready(function(){
      app = new router;
      Backbone.history.start();
    })

  </script>
</head>
<body>
  <div></div>
</body>
</html>
```

Open **index.html** in the browser to see if it works, i.e., the 'Hello World' message should be on the page.

1.2 Working with Collections

The full source code of this example is under rpjs/backbone/collections[5]. It's built on top of "Hello World" example from the **Setting up Backbone.js App from Scratch** exercise which is available for download at rpjs/backbone/hello-world[6].

We should add some data to play around with, and to hydrate our views. To do this, add this right after the **script** tag and before the other code:

```
var appleData = [
  {
    name: "fuji",
    url: "img/fuji.jpg"
  },
  {
    name: "gala",
    url: "img/gala.jpg"
  }
];
```

This is our apple *database*. :-) Or to be more correct, our REST API endpoint-substitute, which provides us with names and image URLs of the apples (data models).

Note

This mock dataset can be easily substituted by assigning REST API endpoints of your back-end to `url` properties in Backbone.js Collections and/or Models, and calling the `fetch()` method on them.

Now to make the User Experience (UX) a little bit better, we can add a new route to the **routes** object in the Backbone Route:

```
...
routes: {
  '': 'home',
  'apples/:appleName': 'loadApple'
},
...
```

This will allow users to go to `index.html#apples/SOMENAME` and expect to see some information about an apple. This information will be fetched and rendered by the **loadApple** function in the Backbone Router definition:

[5] https://github.com/azat-co/rpjs/tree/master/backbone/collections
[6] https://github.com/azat-co/rpjs/tree/master/backbone/hello-world

```
  loadApple: function(appleName){
    this.appleView.render(appleName);
  }
```

Have you noticed an **appleName** variable? It's exactly the same name as the one that we've used in **route**. This is how we can access query string parameters (e.g, `?param=value&q=search`) in Backbone.js.

Now we'll need to refactor some more code to create a Backbone Collection, populate it with data in our **appleData** variable, and to pass the collection to **homeView** and **appleView**. Conveniently enough, we do it all in the Router constructor method **initialize**:

```
  initialize: function(){
    var apples = new Apples();
    apples.reset(appleData);
    this.homeView = new homeView({collection: apples});
    this.appleView = new appleView({collection: apples});
  },
```

At this point, we're pretty much done with the Router class and it should look like this:

```
  var router = Backbone.Router.extend({
    routes: {
      '': 'home',
      'apples/:appleName': 'loadApple'
    },
    initialize: function(){
      var apples = new Apples();
      apples.reset(appleData);
      this.homeView = new homeView({collection: apples});
      this.appleView = new appleView({collection: apples});
    },
    home: function(){
      this.homeView.render();
    },
    loadApple: function(appleName){
      this.appleView.render(appleName);
    }
  });
```

Let's modify our **homeView** a bit to see the whole *database*:

```
var homeView = Backbone.View.extend({
  el: 'body',
  template: _.template('Apple data: <%= data %>'),
  render: function(){
    this.$el.html(this.template({
    data: JSON.stringify(this.collection.models)
  }));
  }
});
```

For now, we just output the string representation of the JSON object in the browser. This is not user-friendly at all, but later we'll improve it by using a list and subviews.

Our apple Backbone Collection is very clean and simple:

```
var Apples = Backbone.Collection.extend({
});
```

Note

Backbone automatically creates models inside of a collection when we use the `fetch()` or `reset()` functions.

Apple view is not any more complex; it has only two properties: **template** and **render**. In a template, we want to display **figure**, **img** and **figcaption** tags with specific values. The Underscore.js template engine is handy at this task:

```
var appleView = Backbone.View.extend({
  template: _.template(
        '<figure>\
           <img src="<%= attributes.url %>"/>\
           <figcaption><%= attributes.name %></figcaption>\
         </figure>'),
...
});
```

To make a JavaScript string, which has HTML tags in it, more readable we can use the backslash line breaker escape (\) symbol, or close strings and concatenate them with a plus sign (+). This is an example of **appleView** above, which is refactored using the latter approach:

```
  var appleView = Backbone.View.extend({
    template: _.template(
        '<figure>'+
          +'<img src="<%= attributes.url %>"/>'+
          +'<figcaption><%= attributes.name %></figcaption>'+
        +'</figure>'),
  ...
```

Please note the '<%=' and '%>' symbols; they are the instructions for Undescore.js to print values in properties **url** and **name** of the **attributes** object.

Finally, we're adding the render function to the **appleView** class.

```
  render: function(appleName){
    var appleModel = this.collection.where({name:appleName})[0];
    var appleHtml = this.template(appleModel);
    $('body').html(appleHtml);
  }
```

We find a model within the collection via `where()` method and use `[]` to pick the first element. Right now, the **render** function is responsible for both loading the data and rendering it. Later we'll refactor the function to separate these two functionalities into different methods.

The whole app, which is in the rpjs/backbone/collections/index.html[7] folder, looks like this:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
   var appleData = [
      {
        name: "fuji",
        url: "img/fuji.jpg"
      },
      {
        name: "gala",
        url: "img/gala.jpg"
      }
    ];
    var app;
```

[7]https://github.com/azat-co/rpjs/tree/master/backbone/collections

```
    var router = Backbone.Router.extend({
      routes: {
        "": "home",
        "apples/:appleName": "loadApple"
      },
      initialize: function(){
        var apples = new Apples();
        apples.reset(appleData);
        this.homeView = new homeView({collection: apples});
        this.appleView = new appleView({collection: apples});
      },
      home: function(){
        this.homeView.render();
      },
      loadApple: function(appleName){
        this.appleView.render(appleName);
      }
    });

    var homeView = Backbone.View.extend({
      el: 'body',
      template: _.template('Apple data: <%= data %>'),
      render: function(){
        this.$el.html(this.template({
        data: JSON.stringify(this.collection.models)
      }));
      }
      //TODO subviews
    });

    var Apples = Backbone.Collection.extend({

    });
    var appleView = Backbone.View.extend({
      template: _.template('<figure>\
                  <img src="<%= attributes.url %>"/>\
                  <figcaption><%= attributes.name %></figcaption>\
                </figure>'),
      //TODO re-write with load apple and event binding
      render: function(appleName){
        var appleModel = this.collection.where({
          name:appleName
        })[0];
        var appleHtml = this.template(appleModel);
        $('body').html(appleHtml);
      }
    });
    $(document).ready(function(){
      app = new router;
      Backbone.history.start();
    })

  </script>
</head>
<body>
  <div></div>
</body>
</html>
```

Open `collections/index.html` file in your browser. You should see the data from our "database", i.e., `Apple data: [{"name":"fuji","url":"img/fuji.jpg"},{"name":"gala","url":"img/gala.jpg"}]`.

Now, let' go to `collections/index.html#apples/fuji` or `collections/index.html#apples/gala` in your browser. We expect to see an image with a caption. It's a detailed view of an item, which in this case is an apple. Nice work!

1.3 Event Binding

In real life, getting data does not happen instantaneously, so let's refactor our code to simulate it. For a better UI/UX, we'll also have to show a loading icon (a.k.a. spinner or ajax-loader) to users to notify them that the information is being loaded.

It's a good thing that we have event binding in Backbone. Without it, we'll have to pass a function that renders HTML as a callback to the data loading function, to make sure that the rendering function is not executed before we have the actual data to display.

Therefore, when a user goes to detailed view (`apples/:id`) we only call the function that loads the data. Then, with the proper event listeners, our view will automagically (this is not a typo) update itself, when there is a new data (or on a data change, Backbone.js supports multiple and even custom events).

Let's change the code in the router:

```
  ...
    loadApple: function(appleName){
      this.appleView.loadApple(appleName);
    }
  ...
```

Everything else remains the same utill we get to the **appleView** class. We'll need to add a constructor or an **initialize** method, which is a special word/property in the Backbone.js framework. It's called each time we create an instance of an object, i.e., `var someObj = new SomeObject()`. We can also pass extra parameters to the **initialize** function, as we did with our views (we passed an object with the key **collection** and the value of **apples** Backbone Collection). Read more on Backbone.js constructors at backbonejs.org/#View-constructor[8].

```
...
var appleView = Backbone.View.extend({
  initialize: function(){
    //TODO: create and setup model (aka an apple)
  },
...
```

Great, we have our **initialize** function. Now we need to create a model which will represent a single apple and set up proper event listeners on the model. We'll use two types of events, `change` and a custom event called `spinner`. To do that, we are going to use the `on()` function, which takes these properties: `on(event, actions, context)` — read more about it at backbonejs.org/#Events-on[9]:

```
...
var appleView = Backbone.View.extend({
    this.model = new (Backbone.Model.extend({}));
    this.model.bind('change', this.render, this);
    this.bind('spinner',this.showSpinner, this);
  },
...
```

The code above basically boils down to two simple things:

1. Call `render()` function of **appleView** object when the model has changed
2. Call `showSpinner()` method of **appleView** object when event **spinner** has been fired.

So far, so good, right? But what about the spinner, a GIF icon? Let's create a new property in **appleView**:

```
...
  templateSpinner: '<img src="img/spinner.gif" width="30"/>',
...
```

Remember the `loadApple` call in the router? This is how we can implement the function in **appleView**:

[8]http://backbonejs.org/#View-constructor
[9]http://backbonejs.org/#Events-on

```
  ...
  loadApple:function(appleName){
    this.trigger('spinner');
    //show spinner GIF image
    var view = this;
    //we'll need to access that inside of a closure
    setTimeout(function(){
    //simulates real time lag when
    //fetching data from the remote server
      view.model.set(view.collection.where({
        name:appleName
      })[0].attributes);
    },1000);
  },
  ...
```

The first line will trigger the `spinner` event (the function for which we still have to write).

The second line is just for scoping issues (so we can use **appleView** inside of the closure).

The `setTimeout` function is simulating a time lag of a real remote server response. Inside of it, we assign attributes of a selected model to our view's model by using a `model.set()` function and a **model.attributes** property (which returns the properties of a model).

Now we can remove an extra code from the `render` method and implement the `showSpinner` function:

```
  render: function(appleName){
    var appleHtml = this.template(this.model);
    $('body').html(appleHtml);
  },
  showSpinner: function(){
    $('body').html(this.templateSpinner);
  }
  ...
```

That's all! Open `index.html#apples/gala` or `index.html#apples/fuji` in your browser and enjoy the loading animation while waiting for an apple image to load.

The full code of the **index.html** file:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
   var appleData = [
      {
        name: "fuji",
        url: "img/fuji.jpg"
      },
      {
        name: "gala",
        url: "img/gala.jpg"
      }
    ];
    var app;
    var router = Backbone.Router.extend({
      routes: {
        "": "home",
        "apples/:appleName": "loadApple"
      },
      initialize: function(){
        var apples = new Apples();
        apples.reset(appleData);
        this.homeView = new homeView({collection: apples});
        this.appleView = new appleView({collection: apples});
      },
      home: function(){
        this.homeView.render();
      },
      loadApple: function(appleName){
        this.appleView.loadApple(appleName);

      }
    });

    var homeView = Backbone.View.extend({
      el: 'body',
      template: _.template('Apple data: <%= data %>'),
      render: function(){
        this.$el.html(this.template({
          data: JSON.stringify(this.collection.models)
        }));
      }
      //TODO subviews
    });

    var Apples = Backbone.Collection.extend({

    });
    var appleView = Backbone.View.extend({
      initialize: function(){
        this.model = new (Backbone.Model.extend({}));
        this.model.on('change', this.render, this);
        this.on('spinner',this.showSpinner, this);
      },
      template: _.template('<figure>\
                <img src="<%= attributes.url %>"/>\
                <figcaption><%= attributes.name %></figcaption>\
                </figure>'),
      templateSpinner: '<img src="img/spinner.gif" width="30"/>',

      loadApple:function(appleName){
        this.trigger('spinner');
        var view = this; //we'll need to access
        //that inside of a closure
        setTimeout(function(){ //simulates real time
        //lag when fetching data from the remote server
          view.model.set(view.collection.where({
            name:appleName
          })[0].attributes);
        },1000);

      },

      render: function(appleName){
        var appleHtml = this.template(this.model);
        $('body').html(appleHtml);
      },
      showSpinner: function(){
        $('body').html(this.templateSpinner);
      }

    });
    $(document).ready(function(){
      app = new router;
      Backbone.history.start();
    })

  </script>
</head>
<body>
  <a href="#apples/fuji">fuji</a>
  <div></div>
</body>
</html>
```

1.4 Views and Subviews with Underscore.js

This example is available at rpjs/backbone/subview[10].

Subviews are Backbone Views that are created and used inside of another Backbone View. A subviews concept is a great way to abstract (separate) UI events (e.g., clicks), and templates for similarly structured elements (e.g., apples).

A use case of a Subview might include a row in a table, a list item in a list, a paragraph, a new line, etc.

We'll refactor our home page to show a nice list of apples. Each list item will have an apple name and a "buy" link with an **onClick** event. Let's start by creating a subview for a single apple with our standard Backbone `extend()` function:

```
  ...
  var appleItemView = Backbone.View.extend({
    tagName: 'li',
    template: _.template(''
           +'<a href="#apples/<%=name%>" target="_blank">'
          +'<%=name%>'
          +'</a>&nbsp;<a class="add-to-cart" href="#">buy</a>'),
    events: {
      'click .add-to-cart': 'addToCart'
    },
    render: function() {
      this.$el.html(this.template(this.model.attributes));
    },
    addToCart: function(){
      this.model.collection.trigger('addToCart', this.model);
    }
  });
  ...
```

Now we can populate the object with **tagName**, **template**, **events**, **render** and **addToCart** properties/methods.

[10]https://github.com/azat-co/rpjs/tree/master/backbone/subview

```
  ...
  tagName: 'li',
  ...
```

tagName automatically allows Backbone.js to create an HTML element with the specified tag name, in this case `<li>` — list item. This will be a representation of a single apple, a row in our list.

```
  ...
  template: _.template(''
         +'<a href="#apples/<%=name%>" target="_blank">'
        +'<%=name%>'
        +'</a>&nbsp;<a class="add-to-cart" href="#">buy</a>'),
  ...
```

The template is just a string with Undescore.js instructions. They are wrapped in `<%` and `%>` symbols. `<%=` simply means print a value. The same code can be written with backslash escapes:

```
  ...
  template: _.template('\
         <a href="#apples/<%=name%>" target="_blank">\
        <%=name%>\
        </a>&nbsp;<a class="add-to-cart" href="#">buy</a>\
        '),
  ...
```

Each `<li>` will have two anchor elements (`<a>`), links to a detailed apple view (`#apples/:appleName`) and a **buy** button. Now we're going to attach an event listener to the **buy** button:

```
  ...
  events: {
    'click .add-to-cart': 'addToCart'
  },
  ...
```

The syntax follows this rule:

```
event + jQuery element selector: function name
```

Both the key and the value (right and left parts separated by the colon) are strings. For example:

```
'click .add-to-cart': 'addToCart'
```

or

```
'click #load-more': 'loadMoreData'
```

To render each item in the list, we'll use the jQuery `html()` function on the `this.$el` jQuery object, which is the `<li>` HTML element based on our **tagName** attribute:

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

addToCart will use the `trigger()` function to notify the collection that this particular model (apple) is up for the purchase by the user:

```
  ...
    addToCart: function(){
      this.model.collection.trigger('addToCart', this.model);
    }
  ...
```

Here is the full code of the **appleItemView** Backbone View class:

```
  ...
  var appleItemView = Backbone.View.extend({
    tagName: 'li',
    template: _.template(''
           +'<a href="#apples/<%=name%>" target="_blank">'
          +'<%=name%>'
          +'</a>&nbsp;<a class="add-to-cart" href="#">buy</a>'),
    events: {
      'click .add-to-cart': 'addToCart'
    },
    render: function() {
      this.$el.html(this.template(this.model.attributes));
    },
    addToCart: function(){
      this.model.collection.trigger('addToCart', this.model);
    }
  });
  ...
```

Easy peasy! But what about the master view, which is supposed to render all of our items (apples) and provide a wrapper `<ul>` container for `<li>` HTML elements? We need to modify and enhance our **homeView**.

To begin with, we can add extra properties of string type understandable by jQuery as selectors to **homeView**:

```
...
el: 'body',
listEl: '.apples-list',
cartEl: '.cart-box',
...
```

We can use properties from above in the template, or just hard-code them (we'll refactor our code later) in **homeView**:

```
...
template: _.template('Apple data: \
  <ul class="apples-list">\
  </ul>\
  <div class="cart-box"></div>'),
...
```

The **initialize** function will be called when **homeView** is created (`new homeView()`) — in it we render our template (with our favorite by now `html()` function), and attach an event listener to the collection (which is a set of apple models):

```
...
  initialize: function() {
    this.$el.html(this.template);
    this.collection.on('addToCart', this.showCart, this);
  },
...
```

The syntax for the binding event is covered in the previous section. In essence, it is calling the `showCart()` function of **homeView**. In this function, we append **appleName** to the cart (along with a line break, a `<br/>` element):

```
...
  showCart: function(appleModel) {
    $(this.cartEl).append(appleModel.attributes.name+'<br/>');
  },
...
```

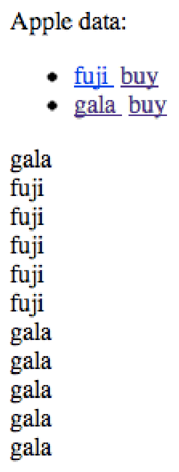
Finally, here is our long-awaited `render()` method, in which we iterate through each model in the collection (each apple), create an **appleItemView** for each apple, create an `<li>` element for each apple, and append that element to **view.listEl** — `<ul>` element with a class apples-list in the DOM:

```
...
render: function(){
  view = this;
  //so we can use view inside of closure
  this.collection.each(function(apple){
    var appleSubView = new appleItemView({model:apple});
    // creates subview with model apple
    appleSubView.render();
    // compiles template and single apple data
    $(view.listEl).append(appleSubView.$el);
    //append jQuery object from single
    //apple to apples-list DOM element
  });
}
...
```

Let’s make sure we didn’t miss anything in the **homeView** Backbone View:

```
...
var homeView = Backbone.View.extend({
  el: 'body',
  listEl: '.apples-list',
  cartEl: '.cart-box',
  template: _.template('Apple data: \
    <ul class="apples-list">\
    </ul>\
    <div class="cart-box"></div>'),
  initialize: function() {
    this.$el.html(this.template);
    this.collection.on('addToCart', this.showCart, this);
  },
  showCart: function(appleModel) {
    $(this.cartEl).append(appleModel.attributes.name+'<br/>');
  },
  render: function(){
    view = this; //so we can use view inside of closure
    this.collection.each(function(apple){
      var appleSubView = new appleItemView({model:apple});
      // create subview with model apple
      appleSubView.render();
      // compiles tempalte and single apple data
      $(view.listEl).append(appleSubView.$el);
      //append jQuery object from single apple
      //to apples-list DOM element
    });
    }
  });
  ...
```

You should be able to click on the buy, and the cart will populate with the apples of your choice. Looking at an individual apple does not require typing its name in the URL address bar of the browser anymore. We can click on the name and it opens a new window with a detailed view.


Apple data:

- fuji buy
- gala buy

```
gala
fuji
fuji
fuji
fuji
fuji
gala
gala
gala
gala
gala
```

The list of apples rendered by subviews.

By using subviews, we reused the template for all of the items (apples) and attached a specific event to each of them. Those events are smart enough to pass the information about the model to other objects: views and collections.

Just in case, here is the full code for the subviews example, which is also available at rpjs/backbone/subview/index.html[11]:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script>
   var appleData = [
      {
        name: "fuji",
        url: "img/fuji.jpg"
```

[11]https://github.com/azat-co/rpjs/blob/master/backbone/subview/index.html

```
    },
    {
      name: "gala",
      url: "img/gala.jpg"
    }
  ];
  var app;
  var router = Backbone.Router.extend({
    routes: {
      "": "home",
      "apples/:appleName": "loadApple"
    },
    initialize: function(){
      var apples = new Apples();
      apples.reset(appleData);
      this.homeView = new homeView({collection: apples});
      this.appleView = new appleView({collection: apples});
    },
    home: function(){
      this.homeView.render();
    },
    loadApple: function(appleName){
      this.appleView.loadApple(appleName);

    }
  });
  var appleItemView = Backbone.View.extend({
    tagName: 'li',
    // template: _.template(''
    //    +'<a href="#apples/<%=name%>" target="_blank">'
    //    +'<%=name%>'
    //    +'</a>&nbsp;<a class="add-to-cart" href="#">buy</a>'),
    template: _.template('\
           <a href="#apples/<%=name%>" target="_blank">\
          <%=name%>\
          </a>&nbsp;<a class="add-to-cart" href="#">buy</a>\
          '),

    events: {
      'click .add-to-cart': 'addToCart'
    },
    render: function() {
      this.$el.html(this.template(this.model.attributes));
    },
    addToCart: function(){
      this.model.collection.trigger('addToCart', this.model);
    }
  });

  var homeView = Backbone.View.extend({
    el: 'body',
    listEl: '.apples-list',
    cartEl: '.cart-box',
    template: _.template('Apple data: \
      <ul class="apples-list">\
      </ul>\
      <div class="cart-box"></div>'),
    initialize: function() {
      this.$el.html(this.template);
      this.collection.on('addToCart', this.showCart, this);
    },
    showCart: function(appleModel) {
      $(this.cartEl).append(appleModel.attributes.name+'<br/>');
    },
    render: function(){
      view = this; //so we can use view inside of closure
      this.collection.each(function(apple){
        var appleSubView = new appleItemView({model:apple});
        // create subview with model apple
        appleSubView.render();
        // compiles tempalte and single apple data
        $(view.listEl).append(appleSubView.$el);
        //append jQuery object from
        //single apple to apples-list DOM element
      });
    }
  });

  var Apples = Backbone.Collection.extend({
  });

  var appleView = Backbone.View.extend({
    initialize: function(){
      this.model = new (Backbone.Model.extend({}));
      this.model.on('change', this.render, this);
      this.on('spinner',this.showSpinner, this);
    },
    template: _.template('<figure>\
                <img src="<%= attributes.url %>"/>\
                <figcaption><%= attributes.name %></figcaption>\
                </figure>'),
      templateSpinner: '<img src="img/spinner.gif" width="30"/>',
      loadApple:function(appleName){
        this.trigger('spinner');
        var view = this;
        //we'll need to access that inside of a closure
        setTimeout(function(){
        //simulates real time lag when fetching data
        // from the remote server
          view.model.set(view.collection.where({
            name:appleName
          })[0].attributes);
        },1000);
      },
      render: function(appleName){
        var appleHtml = this.template(this.model);
        $('body').html(appleHtml);
      },
      showSpinner: function(){
        $('body').html(this.templateSpinner);
      }
    });

    $(document).ready(function(){
      app = new router;
      Backbone.history.start();
    })

  </script>
</head>
<body>
  <div></div>
</body>
</html>
```

The link to an individual item, e.g., `collections/index.html#apples/fuji`, also should work independently, by typing it in the browser address bar.

1.5 Refactoring

At this point you are probably wondering what is the benefit of using the framework and still having multiple classes, objects and elements with different functionalities in one *single* file. This was done for the purpose of adhering to the *Keep it Simple Stupid* (KISS) principle.

The bigger your application is, the more pain there is in unorganized code base. Let's break down our application into multiple files where each file will be one of these types:

- view
- template
- router
- collection
- model

Let's write these scripts to include tags into our **index.html** head — or body, as noted previously:

```
  <script src="apple-item.view.js"></script>
  <script src="apple-home.view.js"></script>
  <script src="apple.view.js"></script>
  <script src="apples.js"></script>
  <script src="apple-app.js"></script>
```

The names don't have to follow the convention of dashes and dots, as long as it's easy to tell what each file is supposed to do.

Now, let's copy our objects/classes into the corresponding files.

Our main **index.html** file should look very minimalistic:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <script src="apple-item.view.js"></script>
  <script src="apple-home.view.js"></script>
  <script src="apple.view.js"></script>
  <script src="apples.js"></script>
  <script src="apple-app.js"></script>

</head>
<body>
  <div></div>
</body>
</html>
```

The other files just have the code that corresponds to their filenames.

The content of **apple-item.view.js**:

```
var appleView = Backbone.View.extend({
  initialize: function(){
    this.model = new (Backbone.Model.extend({}));
    this.model.on('change', this.render, this);
    this.on('spinner',this.showSpinner, this);
  },
  template: _.template('<figure>\
            <img src="<%= attributes.url %>"/>\
            <figcaption><%= attributes.name %></figcaption>\
          </figure>'),
  templateSpinner: '<img src="img/spinner.gif" width="30"/>',

  loadApple:function(appleName){
    this.trigger('spinner');
    var view = this;
    //we'll need to access that inside of a closure
    setTimeout(function(){
    //simulates real time lag when fetching
    //data from the remote server
      view.model.set(view.collection.where({
        name:appleName
      })[0].attributes);
    },1000);

  },

  render: function(appleName){
    var appleHtml = this.template(this.model);
    $('body').html(appleHtml);
  },
  showSpinner: function(){
    $('body').html(this.templateSpinner);
  }

});
```

The **apple-home.view.js** file has the **homeView** object:

```
var homeView = Backbone.View.extend({
  el: 'body',
  listEl: '.apples-list',
  cartEl: '.cart-box',
  template: _.template('Apple data: \
    <ul class="apples-list">\
    </ul>\
    <div class="cart-box"></div>'),
  initialize: function() {
    this.$el.html(this.template);
    this.collection.on('addToCart', this.showCart, this);
  },
  showCart: function(appleModel) {
    $(this.cartEl).append(appleModel.attributes.name+'<br/>');
  },
  render: function(){
    view = this; //so we can use view inside of closure
    this.collection.each(function(apple){
      var appleSubView = new appleItemView({model:apple});
      // create subview with model apple
      appleSubView.render();
      // compiles tempalte and single apple data
      $(view.listEl).append(appleSubView.$el);
      //append jQuery object from
      //single apple to apples-list DOM element
    });
  }
});
```

The **apple.view.js** file contains the master apples' list:

```
var appleView = Backbone.View.extend({
  initialize: function(){
    this.model = new (Backbone.Model.extend({}));
    this.model.on('change', this.render, this);
    this.on('spinner',this.showSpinner, this);
  },
  template: _.template('<figure>\
        <img src="<%= attributes.url %>"/>\
        <figcaption><%= attributes.name %></figcaption>\
      </figure>'),
  templateSpinner: '<img src="img/spinner.gif" width="30"/>',
  loadApple:function(appleName){
    this.trigger('spinner');
    var view = this;
    //we'll need to access that inside of a closure
    setTimeout(function(){
    //simulates real time lag when
    //fetching data from the remote server
      view.model.set(view.collection.where({
        name:appleName
      })[0].attributes);
    },1000);
  },
  render: function(appleName){
    var appleHtml = this.template(this.model);
    $('body').html(appleHtml);
  },
  showSpinner: function(){
    $('body').html(this.templateSpinner);
  }
});
```

apples.js is an empty collection:

```
  var Apples = Backbone.Collection.extend({
  });
```

apple-app.js is the main application file with the data, the router, and the starting command:

```
 var appleData = [
    {
      name: "fuji",
      url: "img/fuji.jpg"
    },
    {
      name: "gala",
      url: "img/gala.jpg"
    }
  ];
  var app;
  var router = Backbone.Router.extend({
    routes: {
      '': 'home',
      'apples/:appleName': 'loadApple'
    },
    initialize: function(){
      var apples = new Apples();
      apples.reset(appleData);
```

```
        this.homeView = new homeView({collection: apples});
        this.appleView = new appleView({collection: apples});
      },
      home: function(){
        this.homeView.render();
      },
      loadApple: function(appleName){
        this.appleView.loadApple(appleName);
      }
    });
    $(document).ready(function(){
      app = new router;
      Backbone.history.start();
    })
```

Now let's try to open the application. It should work exactly the same as in the previous Subviews example.

It's a way better code organization, but it's still far from perfect, because we still have HTML templates directly in the JavaScript code. The problem is that designers and developers can't work on the same files, and any change to the presentation requires a change in the main code base.

We can add a few more JS files to our **index.html** file:

```
  <script src="apple-item.tpl.js"></script>
  <script src="apple-home.tpl.js"></script>
  <script src="apple-spinner.tpl.js"></script>
  <script src="apple.tpl.js"></script>
```

Usually, one Backbone view has one template, but in the case of our **appleView** — detailed view of an apple in a separate window — we also have a spinner, a "loading" GIF animation.

The contents of the files are just global variables which are assigned some string values. Later we can use these variables in our views, when we call the Underscore.js helper method **_.template()**.

The **apple-item.tpl.js** file:

```
var appleItemTpl = '\
     <a href="#apples/<%=name%>" target="_blank">\
    <%=name%>\
    </a>&nbsp;<a class="add-to-cart" href="#">buy</a>\
    ';
```

The **apple-home.tpl.js** file:

```
var appleHomeTpl = 'Apple data: \
        <ul class="apples-list">\
        </ul>\
        <div class="cart-box"></div>';
```

The **apple-spinner.tpl.js** file:

```
var appleSpinnerTpl = '<img src="img/spinner.gif" width="30"/>';
```

The **apple.tpl.js** file:

```
var appleTpl = '<figure>\
                <img src="<%= attributes.url %>"/>\
                <figcaption><%= attributes.name %></figcaption>\
              </figure>';
```

Try to start the application now. The full code is under the rpjs/backbone/refactor[12] folder.

As you can see in the previous example, we used global scoped variables (without the keyword `window`).

Warning

Be careful when you introduce a lot of variables into the global namespace (window keyword). There might be conflicts and other unpredictable consequences. For example, if you wrote an open source library and other developers started using the methods and properties directly, instead of using the interface, what happens later when you decide to finally remove/deprecate those global leaks? To prevent this, properly written libraries and applications use JavaScript closures[13].

Example of using closure and a global variable module definition:

```
(function() {
  var apple= function() {
  ...//do something useful like return apple object
  };
  window.Apple = apple;
}())
```

Or in case when we need to access the app object (which creates a dependency on that object):

[12]https://github.com/azat-co/rpjs/tree/master/backbone/refactor
[13]https://developer.mozilla.org/en-US/docs/JavaScript/Guide/Closures

```
(function() {
  var app = this.app;
  //equivalent of window.appliation
  //in case we need a dependency (app)
  this.apple = function() {
  ...//return apple object/class
  //use app variable
  }
  // eqivalent of window.apple = function(){...};
}())
```

As you can see, we've created the function and called it immediately while also wrapping everything in parentheses `()`.

1.6 AMD and Require.js for Development

AMD allows us to organize development code into modules, manage dependencies, and load them asynchronously. This article does a great job at explaining why AMD is a good thing: WHY AMD?[14]

Start your local HTTP server, e.g., MAMP[15].

Let's enhance our code by using the Require.js library.

Our **index.html** will shrink even more:

```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>
  <script src="require.js"></script>
  <script src="apple-app.js"></script>
</head>
<body>
  <div></div>
</body>
</html>
```

We only included libraries and the single JavaScript file with our application. This file has the following structure:

[14]http://requirejs.org/docs/whyamd.html
[15]http://www.mamp.info/en/index.html

```
require([...],function(...){...});
```

Or in a more explanatory way:

```
require([
  'name-of-the-module',
  ...
  'name-of-the-other-module'
  ],function(referenceToModule, ..., referenceToOtherModule){
  ...//some useful code
  referenceToModule.someMethod();
});
```

Basically, we tell a browser to load the files from the array of filenames — first parameter of the `require()` function — and then pass our modules from those files to the anonymous callback function (second argument) as variables. Inside of the main function (anonymous callback) we can use our modules by referencing those variables. Therefore, our **apple-app.js** metamorphoses into:

```
  require([
    'apple-item.tpl', //can use shim plugin
    'apple-home.tpl',
    'apple-spinner.tpl',
    'apple.tpl',
    'apple-item.view',
    'apple-home.view',
    'apple.view',
    'apples'
  ],function(
    appleItemTpl,
    appleHomeTpl,
    appleSpinnerTpl,
    appleTpl,
    appelItemView,
    homeView,
    appleView,
    Apples
    ){
   var appleData = [
      {
        name: "fuji",
        url: "img/fuji.jpg"
      },
      {
        name: "gala",
        url: "img/gala.jpg"
      }
    ];
    var app;
    var router = Backbone.Router.extend({
    //check if need to be required
      routes: {
        '': 'home',
        'apples/:appleName': 'loadApple'
      },
      initialize: function(){
        var apples = new Apples();
        apples.reset(appleData);
        this.homeView = new homeView({collection: apples});
        this.appleView = new appleView({collection: apples});
      },
      home: function(){
        this.homeView.render();
      },
      loadApple: function(appleName){
        this.appleView.loadApple(appleName);

      }
    });

    $(document).ready(function(){
      app = new router;
      Backbone.history.start();
    })
});
```

We put all of the code inside the function which is a second argument of `require()`, mentioned modules by their filenames, and used dependencies via corresponding parameters. Now we should define the module itself. This is how we can do it with the `define()` method:

```
define([...],function(...){...})
```

The meaning is similar to the `require()` function: dependencies are strings of filenames (and paths) in the array which is passed as the first argument. The second argument is the main function that accepts other libraries as parameters (the order of parameters and modules in the array is important):

```
define(['name-of-the-module'],function(nameOfModule){
  var b = nameOfModule.render();
  return b;
})
```

Note

There is no need to append **.js** to filenames. Require.js does it automatically. Shim plugin is used for importing text files such as HTML templates.

Let’s start with the templates and convert them into the Require.js modules.

The new **apple-item.tpl.js** file:

```
define(function() {
  return '\
             <a href="#apples/<%=name%>" target="_blank">\
            <%=name%>\
            </a>&nbsp;<a class="add-to-cart" href="#">buy</a>\
            '
});
```

The **apple-home.tpl** file:

```
define(function(){
  return 'Apple data: \
        <ul class="apples-list">\
        </ul>\
        <div class="cart-box"></div>';
});
```

The **apple-spinner.tpl.js** file:

```
define(function(){
  return '<img src="img/spinner.gif" width="30"/>';
});
```

The **apple.tpl.js** file:

```
define(function(){
  return '<figure>\
          <img src="<%= attributes.url %>"/>\
          <figcaption><%= attributes.name %></figcaption>\
        </figure>';
 });
```

The **apple-item.view.js** file:

```
define(function() {
  return '\
             <a href="#apples/<%=name%>" target="_blank">\
            <%=name%>\
            </a>&nbsp;<a class="add-to-cart" href="#">buy</a>\
            '
});
```

In the **apple-home.view.js** file, we need to declare dependencies on apple-home.tpl and apple-item.view.js files:

```
define(['apple-home.tpl','apple-item.view'],function(
  appleHomeTpl,
  appleItemView){
return  Backbone.View.extend({
      el: 'body',
      listEl: '.apples-list',
      cartEl: '.cart-box',
      template: _.template(appleHomeTpl),
      initialize: function() {
        this.$el.html(this.template);
        this.collection.on('addToCart', this.showCart, this);
      },
      showCart: function(appleModel) {
        $(this.cartEl).append(appleModel.attributes.name+'<br/>');
      },
      render: function(){
        view = this; //so we can use view inside of closure
        this.collection.each(function(apple){
          var appleSubView = new appleItemView({model:apple});
          // create subview with model apple
          appleSubView.render();
          // compiles tempalte and single apple data
          $(view.listEl).append(appleSubView.$el);
          //append jQuery object from
          //single apple to apples-list DOM element
        });
      }
    });
})
```

The **apple.view.js** file depends on two templates:

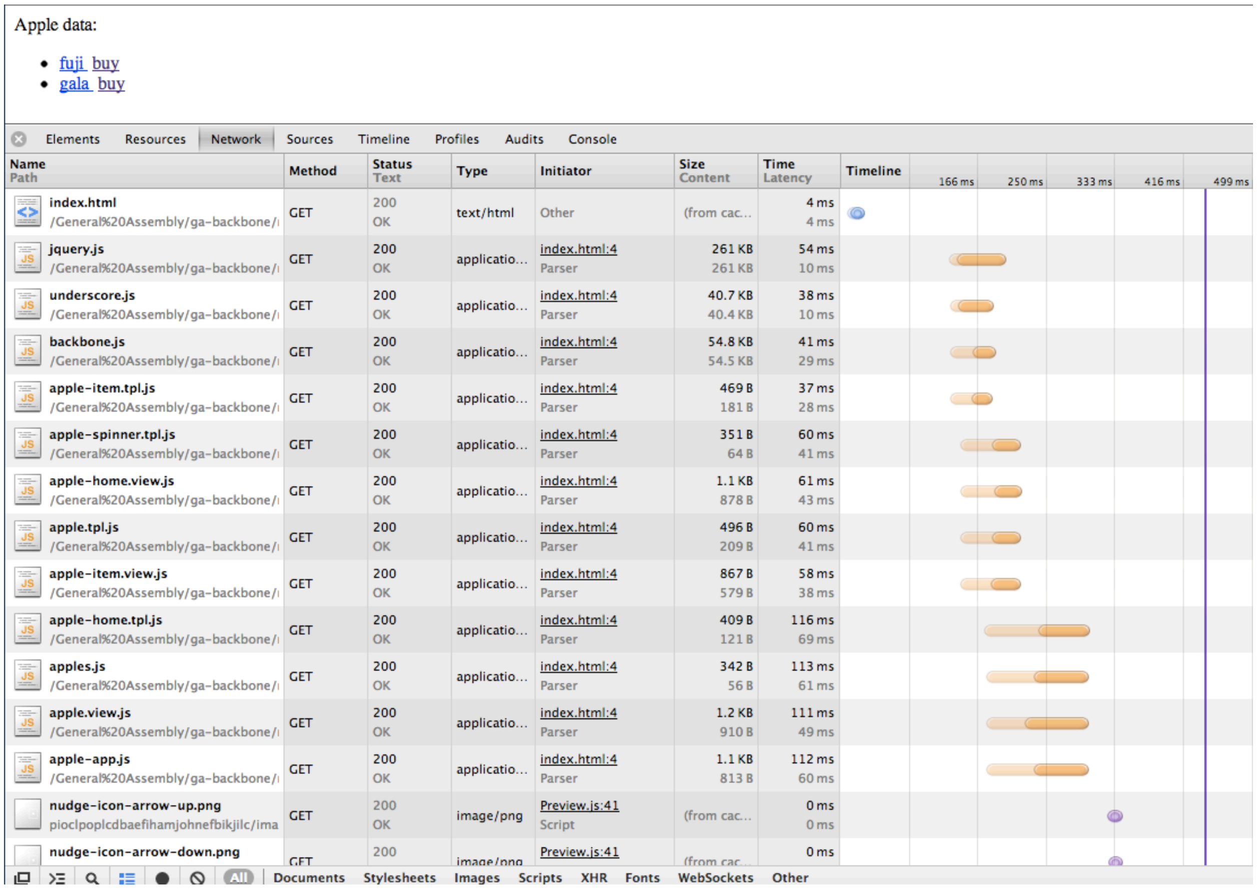
```
define([
  'apple.tpl',
  'apple-spinner.tpl'
],function(appleTpl,appleSpinnerTpl){
  return  Backbone.View.extend({
    initialize: function(){
      this.model = new (Backbone.Model.extend({}));
      this.model.on('change', this.render, this);
      this.on('spinner',this.showSpinner, this);
    },
    template: _.template(appleTpl),
    templateSpinner: appleSpinnerTpl,
    loadApple:function(appleName){
      this.trigger('spinner');
      var view = this;
      //we'll need to access that inside of a closure
      setTimeout(function(){
      //simulates real time lag when
      //fetching data from the remote server
        view.model.set(view.collection.where({
          name:appleName
        })[0].attributes);
      },1000);
    },
    render: function(appleName){
      var appleHtml = this.template(this.model);
      $('body').html(appleHtml);
    },
    showSpinner: function(){
      $('body').html(this.templateSpinner);
    }
  });
});
```

The **apples.js** file:

```
define(function(){
    return Backbone.Collection.extend({})
});
```

I hope you can see the pattern by now. All of our code is split into the separate files based on the logic (e.g., view class, collection class, template). The main file loads all of the dependencies with the `require()` function. If we need some module in a non-main file, then we can ask for it in the `define()` method. Usually, in modules we want to return an object, e.g., in templates we return strings and in views we return Backbone View classes/objects.

Try launching the example under the rpjs/backbone/amd[16] folder. Visually, there shouldn't be any changes. If you open the Network tab in the Developers Tool, you can see a difference in how the files are loaded. The old rpjs/backbone/refactor/index.html[17] file loads our JS scripts in a serial manner while the new the new rpjs/backbone/amd/index.html[18] file loads them in parallel.

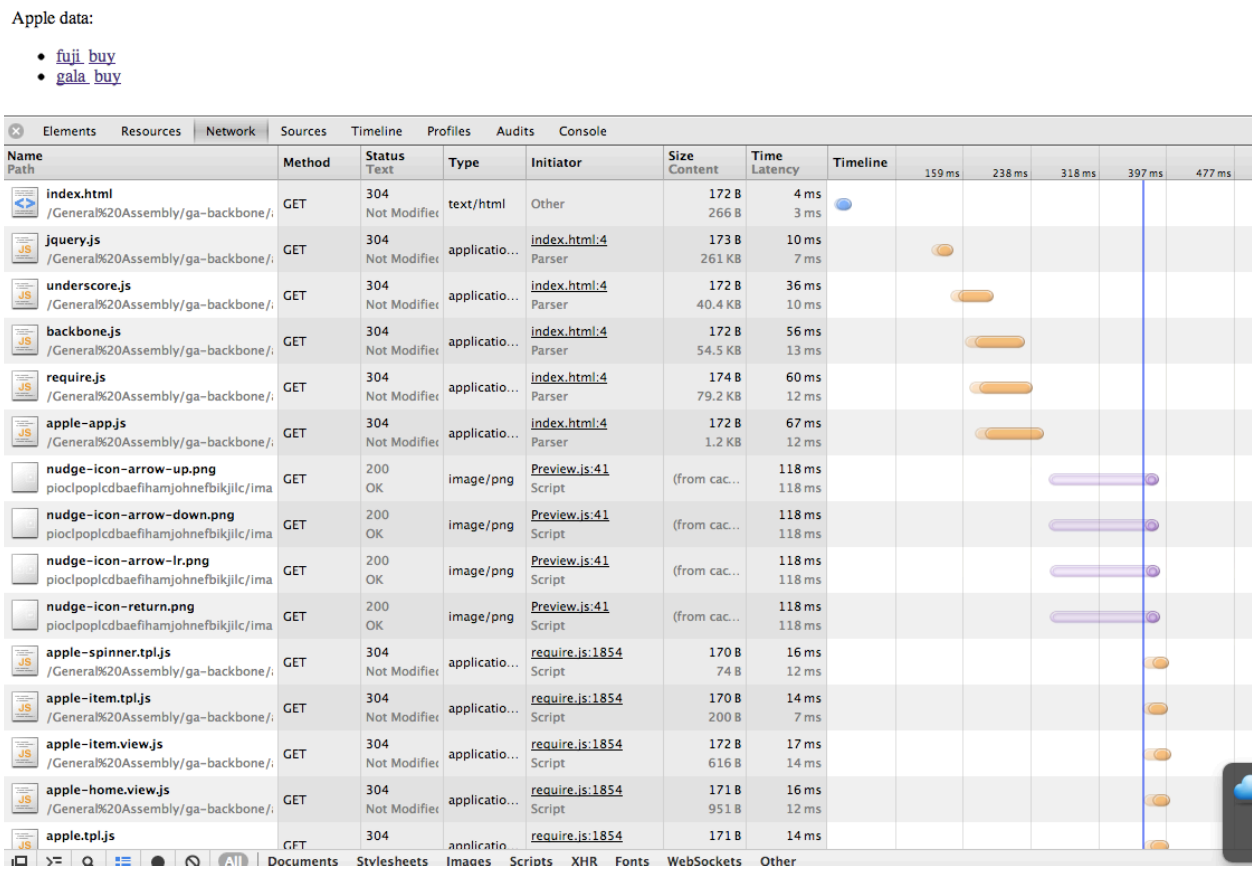

Apple data:

- fuji buy
- gala buy

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency
index.html /General%20Assembly/ga-backbone/	GET	200 OK	text/html	Other	(from cac…	4 ms 4 ms
jquery.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	261 KB 261 KB	54 ms 10 ms
underscore.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	40.7 KB 40.4 KB	38 ms 10 ms
backbone.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	54.8 KB 54.5 KB	41 ms 29 ms
apple-item.tpl.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	469 B 181 B	37 ms 28 ms
apple-spinner.tpl.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	351 B 64 B	60 ms 41 ms
apple-home.view.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	1.1 KB 878 B	61 ms 43 ms
apple.tpl.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	496 B 209 B	60 ms 41 ms
apple-item.view.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	867 B 579 B	58 ms 38 ms
apple-home.tpl.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	409 B 121 B	116 ms 69 ms
apples.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	342 B 56 B	113 ms 61 ms
apple.view.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	1.2 KB 910 B	111 ms 49 ms
apple-app.js /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	index.html:4 Parser	1.1 KB 813 B	112 ms 60 ms
nudge-icon-arrow-up.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	0 ms 0 ms
nudge-icon-arrow-down.png	GET	200	image/png	Preview.js:41	(from cac…	0 ms

The old rpjs/backbone/refactor/index.html file

[16]https://github.com/azat-co/rpjs/tree/master/backbone/amd
[17]https://github.com/azat-co/rpjs/blob/master/backbone/refactor/index.html
[18]https://github.com/azat-co/rpjs/blob/master/backbone/amd/index.html


Apple data:

- fuji buy
- gala buy

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
index.html /General%20Assembly/ga-backbone/	GET	304 Not Modifie	text/html	Other	172 B 266 B	4 ms 3 ms	
jquery.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	index.html:4 Parser	173 B 261 KB	10 ms 7 ms	
underscore.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	index.html:4 Parser	172 B 40.4 KB	36 ms 10 ms	
backbone.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	index.html:4 Parser	172 B 54.5 KB	56 ms 13 ms	
require.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	index.html:4 Parser	174 B 79.2 KB	60 ms 12 ms	
apple-app.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	index.html:4 Parser	172 B 1.2 KB	67 ms 12 ms	
nudge-icon-arrow-up.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	118 ms 118 ms	
nudge-icon-arrow-down.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	118 ms 118 ms	
nudge-icon-arrow-lr.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	118 ms 118 ms	
nudge-icon-return.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	118 ms 118 ms	
apple-spinner.tpl.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	require.js:1854 Script	170 B 74 B	16 ms 12 ms	
apple-item.tpl.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	require.js:1854 Script	170 B 200 B	14 ms 7 ms	
apple-item.view.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	require.js:1854 Script	172 B 616 B	17 ms 14 ms	
apple-home.view.js /General%20Assembly/ga-backbone/	GET	304 Not Modifie	applicatio…	require.js:1854 Script	171 B 951 B	16 ms 12 ms	
apple.tpl.js	GET	304	applicatio	require.js:1854	171 B	14 ms	

The new rpjs/backbone/amd/index.html file

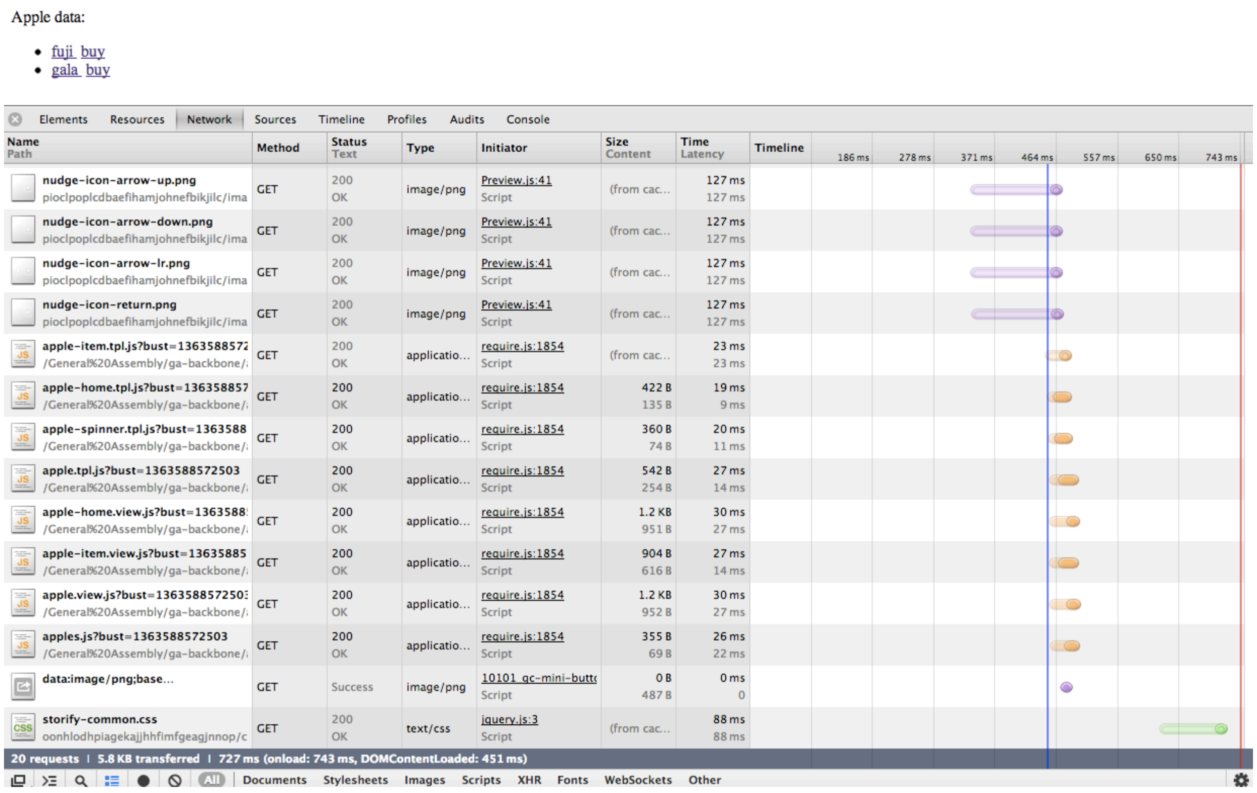
Require.js has a lot of configuration options which are defined through `requirejs.config()` call in a top level of an HTML page. More information can be found at requirejs.org/docs/api.html#config[19].

Let's add a bust parameter to our example. The bust argument will be appended to the URL of each file preventing a browser from caching the files. Perfect for development and terrible for production. :-)

Add this to the **apple-app.js** file in front of everything else:

```
requirejs.config({
  urlArgs: "bust=" +  (new Date()).getTime()
});
require([
...
```

[19]http://requirejs.org/docs/api.html#config



Apple data:

- fuji buy
- gala buy

Elements Resources Network Sources Timeline Profiles Audits Console

Name / Path	Method	Status / Text	Type	Initiator	Size / Content	Time / Latency
nudge-icon-arrow-up.png / pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	127 ms / 127 ms
nudge-icon-arrow-down.png / pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	127 ms / 127 ms
nudge-icon-arrow-lr.png / pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	127 ms / 127 ms
nudge-icon-return.png / pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac…	127 ms / 127 ms
apple-item.tpl.js?bust=1363588572 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	(from cac…	23 ms / 23 ms
apple-home.tpl.js?bust=136358857 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	422 B / 135 B	19 ms / 9 ms
apple-spinner.tpl.js?bust=1363588 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	360 B / 74 B	20 ms / 11 ms
apple.tpl.js?bust=1363588572503 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	542 B / 254 B	27 ms / 14 ms
apple-home.view.js?bust=1363588 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	1.2 KB / 951 B	30 ms / 27 ms
apple-item.view.js?bust=13635885 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	904 B / 616 B	27 ms / 14 ms
apple.view.js?bust=136358857250 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	1.2 KB / 952 B	30 ms / 27 ms
apples.js?bust=1363588572503 / /General%20Assembly/ga-backbone/	GET	200 OK	applicatio…	require.js:1854 Script	355 B / 69 B	26 ms / 22 ms
data:image/png;base…	GET	Success	image/png	10101_qc-mini-butt Script	0 B / 487 B	0 ms / 0
storify-common.css / oonhlodhpiagekajjhhfimfgeagjnnop/c	GET	200 OK	text/css	jquery.js:3 Script	(from cac…	88 ms / 88 ms

Timeline: 186 ms 278 ms 371 ms 464 ms 557 ms 650 ms 743 ms

20 requests | 5.8 KB transferred | 727 ms (onload: 743 ms, DOMContentLoaded: 451 ms)

All Documents Stylesheets Images Scripts XHR Fonts WebSockets Other

Network Tab with bust parameter added

Please note that each file request now has status 200 instead of 304 (not modified).

1.7 Require.js for Production

We'll use the Node Package Manager (NPM) to install the **requirejs** library (it's not a typo; there's no period in the name). In your project folder, run this command in a terminal:

```
$ npm install requirejs
```

Or add `-g` for global installation:

```
$ npm install -g requirejs
```

Create a file **app.build.js**:

```
({
    appDir: "./js",
    baseUrl: "./",
    dir: "build",
    modules: [
        {
            name: "apple-app"
        }
    ]
})
```

Move the script files under the `js` folder (appDir property). The builded files will be placed under the `build` folder (dir parameter). For more information on the build file, check out this *extensive* example with comments: https://github.com/jrburke/r.js/blob/master/build/example.build.js.

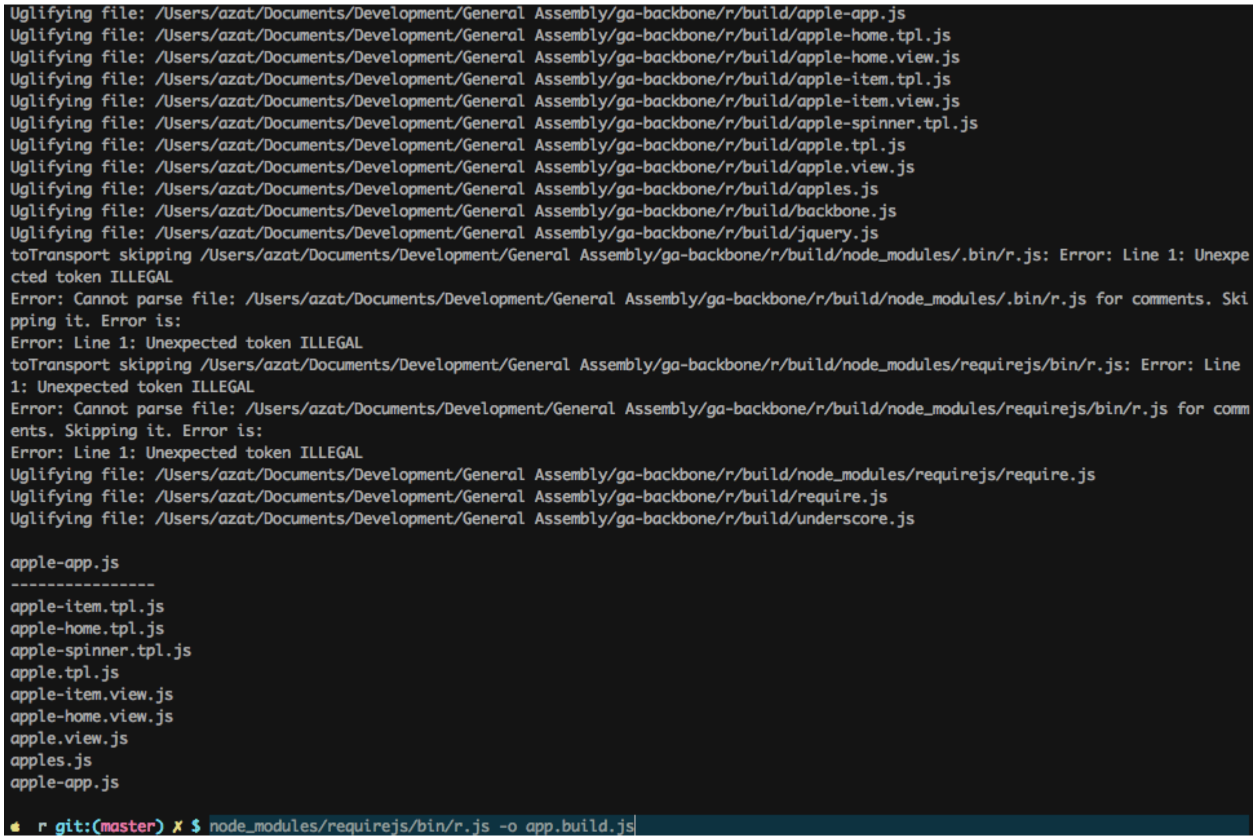
Now everything should be ready for building one gigantic JavaScript file, which will have all of our dependencies/modules:

```
$ r.js -o app.build.js
```

or

```
$ node_modules/requirejs/bin/r.js -o app.build.js
```

You should get a list of the r.js processed files.



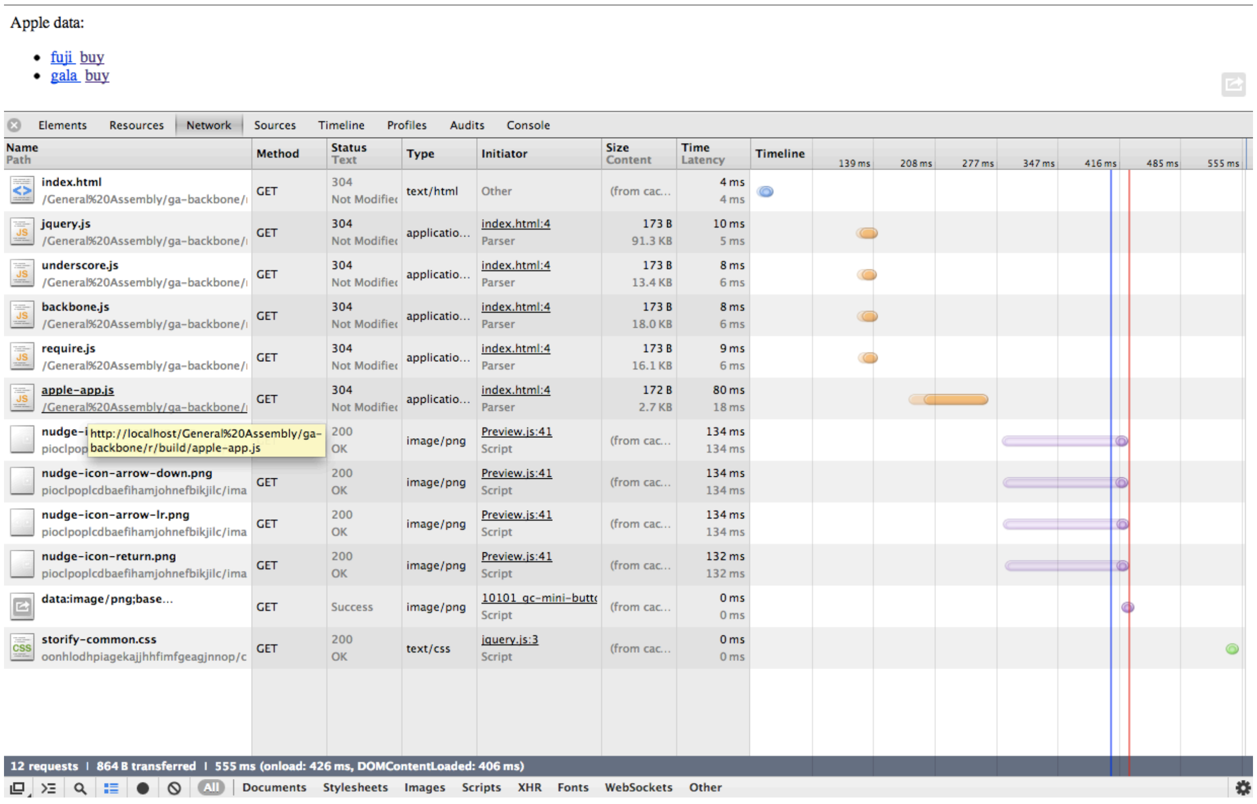
```
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-app.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-home.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-home.view.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-item.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-item.view.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple-spinner.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple.tpl.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apple.view.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/apples.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/backbone.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/jquery.js
toTransport skipping /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/.bin/r.js: Error: Line 1: Unexpe
cted token ILLEGAL
Error: Cannot parse file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/.bin/r.js for comments. Ski
pping it. Error is:
Error: Line 1: Unexpected token ILLEGAL
toTransport skipping /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/requirejs/bin/r.js: Error: Line
1: Unexpected token ILLEGAL
Error: Cannot parse file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/requirejs/bin/r.js for comm
ents. Skipping it. Error is:
Error: Line 1: Unexpected token ILLEGAL
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/node_modules/requirejs/require.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/require.js
Uglifying file: /Users/azat/Documents/Development/General Assembly/ga-backbone/r/build/underscore.js

apple-app.js
----------------
apple-item.tpl.js
apple-home.tpl.js
apple-spinner.tpl.js
apple.tpl.js
apple-item.view.js
apple-home.view.js
apple.view.js
apples.js
apple-app.js

r git:(master) ✗ $ node_modules/requirejs/bin/r.js -o app.build.js
```

A list of the r.js processed files.

Open **index.html** from the build folder in a browser window, and check if the Network Tab shows any improvement now with just one request/file to load.



Apple data:

- fuji buy
- gala buy

Name Path	Method	Status Text	Type	Initiator	Size Content	Time Latency	Timeline
index.html /General%20Assembly/ga-backbone/	GET	304 Not Modified	text/html	Other	(from cac...	4 ms 4 ms	
jquery.js /General%20Assembly/ga-backbone/	GET	304 Not Modified	applicatio...	index.html:4 Parser	173 B 91.3 KB	10 ms 5 ms	
underscore.js /General%20Assembly/ga-backbone/	GET	304 Not Modified	applicatio...	index.html:4 Parser	173 B 13.4 KB	8 ms 6 ms	
backbone.js /General%20Assembly/ga-backbone/	GET	304 Not Modified	applicatio...	index.html:4 Parser	173 B 18.0 KB	8 ms 6 ms	
require.js /General%20Assembly/ga-backbone/	GET	304 Not Modified	applicatio...	index.html:4 Parser	173 B 16.1 KB	9 ms 6 ms	
apple-app.js /General%20Assembly/ga-backbone/	GET	304 Not Modified	applicatio...	index.html:4 Parser	172 B 2.7 KB	80 ms 18 ms	
nudge-i pioclpop		200 OK	image/png	Preview.js:41 Script	(from cac...	134 ms 134 ms	
nudge-icon-arrow-down.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac...	134 ms 134 ms	
nudge-icon-arrow-lr.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac...	134 ms 134 ms	
nudge-icon-return.png pioclpoplcdbaefihamjohnefbikjilc/ima	GET	200 OK	image/png	Preview.js:41 Script	(from cac...	132 ms 132 ms	
data:image/png;base...	GET	Success	image/png	10101_gc-mini-butt Script	(from cac...	0 ms 0 ms	
storify-common.css oonhlodhpiagekajjhhfimfgeagjnnop/c	GET	200 OK	text/css	jquery.js:3 Script	(from cac...	0 ms 0 ms	

http://localhost/General%20Assembly/ga-backbone/r/build/apple-app.js

12 requests | 864 B transferred | 555 ms (onload: 426 ms, DOMContentLoaded: 406 ms)

Performance improvement with one request/file to load.

For more information, check out the official r.js documentation at requirejs.org/docs/optimization.html[20].

The example code is available under the rpjs/backbone/r[21] and rpjs/backbone/r/build[22] folders.

For uglification of JS files (decreases the files' sizes), we can use the Uglify2[23] module. To install it with NPM, use:

```
$ npm install uglify-js
```

Then update the **app.build.js** file with the `optimize: "uglify2"` property:

[20]http://requirejs.org/docs/optimization.html
[21]https://github.com/azat-co/rpjs/tree/master/backbone/r
[22]https://github.com/azat-co/rpjs/tree/master/backbone/r/build
[23]https://github.com/mishoo/UglifyJS2

```
({
    appDir: "./js",
    baseUrl: "./",
    dir: "build",
    optimize: "uglify2",
    modules: [
        {
            name: "apple-app"
        }
    ]
})
```

Run r.js with:

```
$ node_modules/requirejs/bin/r.js -o app.build.js
```

You should get something like this:

```
define("apple-item.tpl",[],function(){return'             <a href="#apples/<%=name%>" targ\
et="_blank">            <%=name%>            </a>&nbsp;<a class="add-to-cart" href="#">buy\
</a>            '}),define("apple-home.tpl",[],function(){return'Apple data:         <ul c\
lass="apples-list">        </ul>        <div class="cart-box"></div>'}),define("apple-spin\
ner.tpl",[],function(){return'<img src="img/spinner.gif" width="30"/>'}),define("apple.tpl\
",[],function(){return'<figure>                              <img src="<%= attributes.url \
%>"/>                              <figcaption><%= attributes.name %></figcaption>        \
                    </figure>'}),define("apple-item.view",["apple-item.tpl"],function(e){r\
eturn Backbone.View.extend({tagName:"li",template:_.template(e),events:{"click .add-to-car\
t":"addToCart"},render:function(){this.$el.html(this.template(this.model.attributes))},add\
ToCart:function(){this.model.collection.trigger("addToCart",this.model)}})}),define("apple\
-home.view",["apple-home.tpl","apple-item.view"],function(e,t){return Backbone.View.extend\
({el:"body",listEl:".apples-list",cartEl:".cart-box",template:_.template(e),initialize:fun\
ction(){this.$el.html(this.template),this.collection.on("addToCart",this.showCart,this)},s\
howCart:function(e){$(this.cartEl).append(e.attributes.name+"<br/>")},render:function(){vi\
ew=this,this.collection.each(function(e){var i=new t({model:e});i.render(),$(view.listEl).\
append(i.$el)})}})}),define("apple.view",["apple.tpl","apple-spinner.tpl"],function(e,t){r\
eturn Backbone.View.extend({initialize:function(){this.model=new(Backbone.Model.extend({})\
),this.model.on("change",this.render,this),this.on("spinner",this.showSpinner,this)},templ\
ate:_.template(e),templateSpinner:t,loadApple:function(e){this.trigger("spinner");var t=th\
is;setTimeout(function(){t.model.set(t.collection.where({name:e})[0].attributes)},1e3)},re\
nder:function(){var e=this.template(this.model);$("body").html(e)},showSpinner:function(){\
$("body").html(this.templateSpinner)}})}),define("apples",[],function(){return Backbone.Co\
llection.extend({})}),requirejs.config({urlArgs:"bust="+(new Date).getTime()}),require(["a\
pple-item.tpl","apple-home.tpl","apple-spinner.tpl","apple.tpl","apple-item.view","apple-h\
ome.view","apple.view","apples"],function(e,t,i,n,a,l,p,o){var r,s=[{name:"fuji",url:"img/\
fuji.jpg"},{name:"gala",url:"img/gala.jpg"}],c=Backbone.Router.extend({routes:{"":"home","\
apples/:appleName":"loadApple"},initialize:function(){var e=new o;e.reset(s),this.homeView\
=new l({collection:e}),this.appleView=new p({collection:e})},home:function(){this.homeView\
.render()},loadApple:function(e){this.appleView.loadApple(e)}});$(document).ready(function\
(){r=new c,Backbone.history.start()})}),define("apple-app",function(){});
```

Note

The file is not formatted on purpose to show how Uglify2 works. Without the line break escape symbols, the code is on one line. Also notice that variables and objects' names are shortened.

1.8 Super Simple Backbone Starter Kit

To jump-start your Backbone.js development, consider using Super Simple Backbone Starter Kit[24] or similar projects:

- Backbone Boilerplate[25]
- Sample App with Backbone.js and Twitter Bootstrap[26]
- More Backbone.js tutorials github.com/documentcloud/backbone/wiki/Tutorials%2C-blog-posts-and-example-sites[27].

[24]https://github.com/azat-co/super-simple-backbone-starter-kit
[25]http://backboneboilerplate.com/
[26]http://coenraets.org/blog/2012/02/sample-app-with-backbone-js-and-twitter-bootstrap/
[27]https://github.com/documentcloud/backbone/wiki/Tutorials%2C-blog-posts-and-example-sites

2. BONUS: Webapplog Articles

Summary: articles on the essence of asynchronocity in Node.js, TDD with Mocha; introduction to Express.js, Monk, Wintersmith, Derby frameworks/libraries.

"Don't worry about failure; you only have to be right once." — Drew Houston[1]

For your convenience we included some of the Node.js posts from Webapplog.com[2] — a publicly accessible blog about web development — in this chapter.

2.1 Asynchronicity in Node

2.1.1 Non-Blocking I/O

One of the biggest advantages of using Node.js over Python or Ruby is that Node has a non-blocking I/O mechanism. To illustrate this, let me use an example of a line in a Starbucks coffeeshop. Let's pretend that each person standing in line for a drink is a task, and everything behind the counter — cashier, register, barista — is a server or server application. Whether we order a cup of regular drip coffee, like Pike Place, or hot tea, like Earl Grey, the barista makes it. The whole line waits while that drink is made, and each person is charged the appropriate amount.

Of course, we know the aforementioned drinks (a.k.a., time-consuming bottlenecks) are easy to make; just pour the liquid and it's done. But what about those fancy choco-mocha-frappe-latte-soy-decafs? What if everybody in line decides to order these time-consuming drinks? The line will be held up, and in turn, grow longer and longer. The manager of the coffeeshop will have to add more registers and put more baristas to work (or even stand behind the register him/herself).

This is not good, right? But this is how virtually all server-side technologies work, except Node.js, which is like a real Starbucks. When you order something, the barista yells the order to the other employee, and you leave the register. Another person gives their order while you wait for your state-of-the-art eye-opener in a paper cup. The line moves, the processes are executed asynchronously and without blocking the queue.

This is why Node.js blows everything else away (except maybe low-level C++) in terms of performance and scalability. With Node.js, you just don't need that many CPUs and servers to handle the load.

[1]http://en.wikipedia.org/wiki/Drew_Houston
[2]http://webapplog.com

2.1.2 Asynchronous Way of Coding

Asynchronicity requires a different way of thinking for programmers familiar with Python, PHP, C or Ruby. It's easy to introduce a bug unintentionally by forgetting to end the execution of the code with a proper **return** expression.

Here is a simple example illustrating this scenario:

```
var test = function (callback) {
  return callback();
  console.log('test') //shouldn't be printed
}

var test2 = function(callback){
  callback();
  console.log('test2') //printed 3rd
}

test(function(){
  console.log('callback1') //printed first
  test2(function(){
  console.log('callback2') //printed 2nd
  })
});
```

If we don't use return callback() and just use callback() our string test2 will be printed (test is not printed).

```
callback1
callback2
tes2
```

For fun I've added a `setTimeout()` delay for the callback2 string, and now the order has changed:

```
var test = function (callback) {
  return callback();
  console.log('test') //shouldn't be printed
}

var test2 = function(callback){
  callback();
  console.log('test2') //printed 2nd
}

test(function(){
  console.log('callback1') //printed first
  test2(function(){
    setTimeout(function(){
      console.log('callback2') //printed 3rd
    },100)
  })
});
```

Prints:

```
callback1
tes2
callback2
```

The last example illustrates that the two functions are independent of each other and run in parallel. The faster function will finish sooner than the slower one. Going back to our Starbucks examples, you might get your drink faster than the other person who was in front of you in the line. Better for people, and better for programs! :-)

2.2 MongoDB Migration with Monk

Recently one of our top users complained that his Storify[3] account was inaccessible. We've checked the production database, and it appears that the account might have been compromised and maliciously deleted by somebody using the user's account credentials. Thanks to a great MongoHQ service, we had a backup database in less than 15 minutes. There were two options to proceed with the migration:

1. Mongo shell script
2. Node.js program

Because Storify user account deletion involves deletion of all related objects — identities, relationships (followers, subscriptions), likes, stories — we've decided to proceed with the latter option. It worked perfectly, and here is a simplified version which you can use as a boilerplate for MongoDB migration (also at gist.github.com/4516139[4]).

Let's load all of the modules we need: Monk[5], Progress[6], Async[7], and MongoDB:

[3]http://storify.com
[4]https://gist.github.com/4516139
[5]https://github.com/LearnBoost/monk
[6]https://github.com/visionmedia/node-progress
[7]https://github.com/caolan/async

```
var async = require('async');
var ProgressBar = require('progress');
var monk = require('monk');
var ObjectId=require('mongodb').ObjectID;
```

By the way, Monk, made by LeanBoost[8], is a “tiny layer that provides simple yet substantial usability improvements for MongoDB usage within Node.js”.

Monk takes connection string in the following format:

```
username:password@dbhost:port/database
```

So we can create the following objects:

```
var dest = monk('localhost:27017/storify_localhost');
var backup = monk('localhost:27017/storify_backup');
```

We need to know the object ID which we want to restore:

```
var userId = ObjectId(YOUR-OBJECT-ID);
```

This is a handy `restore()` function which we can reuse to restore objects from related collections by specifying the query (for more on MongoDB queries, go to the post Querying 20M-Record MongoDB Collection[9]). To call it, just pass a name of the collection as a string, e.g., `"stories"` and a query which associates objects from this collection with your main object, e.g., `{userId:user.id}`. The progress bar is needed to show us nice visuals in the terminal:

```
var restore = function(collection, query, callback){
  console.info('restoring from ' + collection);
  var q = query;
  backup.get(collection).count(q, function(e, n) {
    console.log('found '+n+' '+collection);
    if (e) console.error(e);
    var bar = new ProgressBar('[:bar] :current/:total'
      + ':percent :etas'
      , { total: n-1, width: 40 })
    var tick = function(e) {
      if (e) {
        console.error(e);
        bar.tick();
      }
      else {
```

[8]https://www.learnboost.com/

[9]http://www.webapplog.com/querying-20m-record-mongodb-collection/

```
        bar.tick();
      }
      if (bar.complete) {
        console.log();
        console.log('restoring '+collection+' is completed');
        callback();
      }
    };
    if (n>0){
      console.log('adding '+ n+ ' '+collection);
      backup.get(collection).find(q, {
        stream: true
      }).each(function(element) {
        dest.get(collection).insert(element, tick);
      });
    } else {
      callback();
    }
  });
}
```

Now we can use async to call the `restore()` function mentioned above:

```
async.series({
  restoreUser: function(callback){   // import user element
    backup.get('users').find({_id:userId}, {
      stream: true, limit: 1
    }).each(function(user) {
      dest.get('users').insert(user, function(e){
        if (e) {
          console.log(e);
        }
        else {
          console.log('resored user: '+ user.username);
        }
        callback();
      });
    });
  },

  restoreIdentity: function(callback){
    restore('identities',{
      userid:userId
    }, callback);
  },

  restoreStories: function(callback){
    restore('stories', {authorid:userId}, callback);
  }

  }, function(e) {
  console.log();
  console.log('restoring is completed!');
  process.exit(1);
});
```

The full code is available at gist.github.com/4516139[10] and here:

```
var async = require('async');
var ProgressBar = require('progress');
var monk = require('monk');
var ms = require('ms');
var ObjectId=require('mongodb').ObjectID;

var dest = monk('localhost:27017/storify_localhost');
var backup = monk('localhost:27017/storify_backup');

var userId = ObjectId(YOUR-OBJECT-ID);
// monk should have auto casting but we need it for queries

var restore = function(collection, query, callback){
  console.info('restoring from ' + collection);
  var q = query;
  backup.get(collection).count(q, function(e, n) {
    console.log('found '+n+' '+collection);
    if (e) console.error(e);
    var bar = new ProgressBar(
      '[:bar] :current/:total :percent :etas',
      { total: n-1, width: 40 })
    var tick = function(e) {
      if (e) {
        console.error(e);
        bar.tick();
      }
      else {
        bar.tick();
      }
      if (bar.complete) {
        console.log();
```

[10] https://gist.github.com/4516139

```
        console.log('restoring '+collection+' is completed');
        callback();
      }
    };
    if (n>0){
      console.log('adding '+ n+ ' '+collection);
      backup.get(collection).find(q, { stream: true })
        .each(function(element) {
        dest.get(collection).insert(element, tick);
      });
    } else {
      callback();
    }
  });
}

async.series({
  restoreUser: function(callback){   // import user element
    backup.get('users').find({_id:userId}, {
      stream: true,
      limit: 1 })
      .each(function(user) {
      dest.get('users').insert(user, function(e){
        if (e) {
          console.log(e);
        }
        else {
          console.log('resored user: '+ user.username);
        }
        callback();
      });
    });
  },

  restoreIdentity: function(callback){
    restore('identities',{
      userid:userId
    }, callback);
  },

  restoreStories: function(callback){
    restore('stories', {authorid:userId}, callback);
  }

  }, function(e) {
```

```
  console.log();
  console.log('restoring is completed!');
  process.exit(1);
});
```

To launch it, run `npm install`/`npm update` and change the hard-coded database values.

2.3 TDD in Node.js with Mocha

2.3.1 Who Needs Test-Driven Development?

Imagine that you need to implement a complex feature on top of an existing interface, e.g., a ‘like’ button on a comment. Without tests, you’ll have to manually create a user, log in, create a post, create a different user, log in with a different user and like the post. Tiresome? What if you’ll need to do it 10 or 20 times to find and fix some nasty bug? What if your feature breaks existing functionality, but you notice it six months after the release because there was no test?!

Don’t waste time writing tests for throwaway scripts, but please adapt the habit of Test-Driven Development for the main code base. With a little time spent in the beginning, you and your team will save time later and have confidence when rolling out new releases. Test Driven Development is a really, really, really good thing.

2.3.2 Quick Start Guide

Follow this quick guide to set up your Test-Driven Development process in Node.js with Mocha[11].

Install Mocha[12] globally by executing this command:

```
$ sudo npm install -g mocha
```

We’ll also use two libraries, Superagent[13] and expect.js[14] by LearnBoost[15]. To install them, fire up NPM[16] commands in your project folder like this:

```
$ npm install superagent
$ npm install expect.js
```

Open a new file with `.js` extension and type:

[11]http://visionmedia.github.com/mocha/
[12]http://visionmedia.github.com/mocha/
[13]https://github.com/visionmedia/superagent
[14]https://github.com/LearnBoost/expect.js/
[15]https://github.com/LearnBoost
[16]https://npmjs.org/

```
var request = require('superagent');
var expect = require('expect.js');
```

So far we've included two libraries. The structure of the test suite going to look like this:

```
describe('Suite one', function(){
  it(function(done){
  ...
  });
  it(function(done){
  ...
  });
});
describe('Suite two', function(){
  it(function(done){
  ...
  });
});
```

Inside of this closure, we can write a request to our server, which should be running at localhost:8080[17]:

```
...
it (function(done){
  request.post('localhost:8080').end(function(res){
    //TODO check that response is okay
  });
});
...
```

Expect will give us handy functions to check any condition we can think of:

```
...
expect(res).to.exist;
expect(res.status).to.equal(200);
expect(res.body).to.contain('world');
...
```

Lastly, we need to add the **done()** call to notify Mocha that the asynchronous test has finished its work. And the full code of our first test looks like this:

[17]http://localhost:8080

```
var request = require('superagent');
var expect = require('expect.js');

describe('Suite one', function(){
 it (function(done){
   request.post('localhost:8080').end(function(res){
    expect(res).to.exist;
    expect(res.status).to.equal(200);
    expect(res.body).to.contain('world');
    done();
  });
});
```

If we want to get fancy, we can add **before** and **beforeEach** hooks which will, according to their names, execute once before the test (or suite) or each time before the test (or suite):

```
before(function(){
  //TODO seed the database
});
describe('suite one ',function(){
  beforeEach(function(){
    //todo log in test user
  });
  it('test one', function(done){
  ...
  });
});
```

Note that before and beforeEach can be placed inside or outside of the describe construction.

To run our test, simply execute:

```
$ mocha test.js
```

To use a different report type:

```
$ mocha test.js -R list
$ mocah test.js -R spec
```

2.4 Wintersmith — Static Site Generator

For this book's one-page website —rapidprototypingwithjs.com[18] — I used Wintersmith[19] to learn something new and to ship fast. Wintersmith is a Node.js static site generator. It greatly impressed me with its flexibility

[18]http://rapidprototypingwithjs.com
[19]http://jnordberg.github.com/wintersmith/

and ease of development. In addition, I could stick to my favorite tools such as Markdown[20], Jade and Underscore[21].

Why Static Site Generators

Here is a good article on why using a static site generator is a good idea in general: An Introduction to Static Site Generators[22]. It basically boils down to a few main things:

Templates

You can use template engines such as Jade[23]. Jade uses whitespaces to structure nested elements, and its syntax is similar to Ruby on Rail's Haml markup.

Markdown

I've copied markdown text from my book's Introduction chapter and used it without any modifications. Wintersmith comes with a marked[24] parser by default. More on why Markdown is great in my old post: Markdown Goodness[25].

Simple Deployment

Everything is HTML, CSS and JavaScript so you just upload the files with an FTP client, e.g., Transmit[26] (by Panic) or Cyberduck[27].

Basic Hosting

Due to the fact that any static web server will work well, there is no need for Heroku or Nodejitsu PaaS solutions, or even PHP/MySQL hosting.

Performance

There are no database calls, no server-side API calls, and no CPU/RAM overhead.

Flexibility

Wintersmith allows for different plugins for contents and templates, and you can even write your own plugins[28].

2.4.1 Getting Started with Wintersmith

There is a quick getting started guide on github.com/jnordberg/wintersmith[29].

To install Wintersmith globally, run NPM with -g and sudo:

[20]http://daringfireball.net/projects/markdown/
[21]http://underscorejs.org/
[22]http://www.mickgardner.com/2012/12/an-introduction-to-static-site.html
[23]https://github.com/visionmedia/jade
[24]https://github.com/chjj/marked
[25]http://www.webapplog.com/markdown-goodness/
[26]http://www.panic.com/transmit/
[27]http://cyberduck.ch/
[28]https://github.com/jnordberg/wintersmith#content-plugins
[29]https://github.com/jnordberg/wintersmith

```
$ sudo npm install wintersmith -g
```

Then run to use the default blog template:

```
$ wintersmith new <path>
```

or for an empty site:

```
$ wintersmith new <path> -template basic
```

or use a shortcut:

```
$ wintersmith new <path> -T basic
```

Similar to Ruby on Rails scaffolding, Wintersmith will generate a basic skeleton with **contents** and **templates** folders. To preview a website, run these commands:

```
$ cd <path>
$ wintersmith preview
$ open http://localhost:8080
```

Most of the changes will be updates automatically in the preview mode, except for the config.json file[30].

Images, CSS, JavaScript and other files go into the **contents** folder. The Wintersmith generator has the following logic:

1. looks for *.md files in the contents folder
2. reads metadata[31] such as the template name
3. processes *.jade templates[32] per metadata in *.md files

When you're done with your static site, just run:

```
$ wintersmith build
```

[30]https://github.com/jnordberg/wintersmith#config
[31]https://github.com/jnordberg/wintersmith#the-page-plugin
[32]https://github.com/jnordberg/wintersmith#templates

2.4.2 Other Static Site Generators

Here are some of the other Node.js static site generators:

- DocPad[33]
- Blacksmith[34]
- Scotch[35]
- Wheat[36]
- Petrify[37]

A more detailed overview of these static site generators is available in the post Node.js Based Static Site Generators[38].

For other languages and frameworks like Rails and PHP, take a look at Static Site Generators by GitHub Watcher Count[39] and the "mother of all site generator lists[40]".

2.5 Intro to Express.js: Simple REST API app with Monk and MongoDB

2.5.1 REST API app with Express.js and Monk

This app is a start of a mongoui[41] project — a phpMyAdmin counterpart for MongoDB written in Node.js. The goal is to provide a module with a nice web admin user interface. It will be something like Parse.com[42], Firebase.com[43], MongoHQ[44] or MongoLab[45] has, but without tying it to any particular service. Why do we have to type `db.users.findOne({'_id':ObjectId('...')})` anytime we want to look up the user information? The alternative MongoHub[46] Mac app is nice (and free) but clunky to use and not web-based.

Ruby enthusiasts like to compare Express to the Sinatra[47] framework. It's similarly flexible in the way developers can build their apps. Application routes are set up in a similar manner, i.e., `app.get('/products/:id', showProduct);`. Currently Express.js is at version number 3.1. In addition to Express, we'll use the Monk[48] module.

We'll use Node Package Manager[49], which usually comes with a Node.js installation. If you don't have it

[33]https://github.com/bevry/docpad#readme
[34]https://github.com/flatiron/blacksmith
[35]https://github.com/techwraith/scotch
[36]https://github.com/creationix/wheat
[37]https://github.com/caolan/petrify
[38]http://blog.bmannconsulting.com/node-static-site-generators/
[39]https://gist.github.com/2254924
[40]http://nanoc.stoneship.org/docs/1-introduction/#similar-projects
[41]http://gitbhub.com/azat-co/mongoui
[42]http://parse.com
[43]http://firebase.com
[44]http://mongohq.com
[45]http://mongolab.com
[46]http://mongohub.todayclose.com/
[47]http://www.sinatrarb.com/
[48]https://github.com/LearnBoost/monk
[49]http://npmjs.org

already, you can get it at npmjs.org[50].

Create a new folder and NPM configuration file, **package.json**, in it with the following content:

```
{
  "name": "mongoui",
  "version": "0.0.1",
  "engines": {
    "node": ">= v0.6"
  },
  "dependencies": {
    "mongodb":"1.2.14",
    "monk": "0.7.1",
    "express": "3.1.0"
  }
}
```

Now run `npm install` to download and install modules into the **node_module** folder. If everything went okay you'll see bunch of folders in **node_modules** folders. All of the code for our application will be in one file, **index.js**, to keep it simple stupid:

```
var mongo = require('mongodb');
var express = require('express');
var monk = require('monk');
var db =  monk('localhost:27017/test');
var app = new express();

app.use(express.static(__dirname + '/public'));
app.get('/',function(req,res){
  db.driver.admin.listDatabases(function(e,dbs){
      res.json(dbs);
  });
});
app.get('/collections',function(req,res){
  db.driver.collectionNames(function(e,names){
    res.json(names);
  })
});
app.get('/collections/:name',function(req,res){
  var collection = db.get(req.params.name);
  collection.find({},{limit:20},function(e,docs){
    res.json(docs);
  })
});
app.listen(3000)
```

[50] http://npmjs.org

Let's break down the code piece by piece. Module declaration:

```
var mongo = require('mongodb');
var express = require('express');
var monk = require('monk');
```

Database and Express application instantiation:

```
var db =  monk('localhost:27017/test');
var app = new express();
```

Tell Express application to load and server static files (if there are any) from the public folder:

```
app.use(express.static(__dirname + '/public'));
```

Home page, a.k.a. root route, set up:

```
app.get('/',function(req,res){
  db.driver.admin.listDatabases(function(e,dbs){
      res.json(dbs);
  });
});
```

`get()` function just takes two parameters: string and function. The string can have slashes and colons — for example, `product/:id`. The function must have two parameters: request and response. Request has all of the information like query string parameters, session and headers, and response is an object to which we output the results. In this case, we do it by calling the `res.json()` function.

`db.driver.admin.listDatabases()`, as you might guess, gives us a list of databases in an asynchronous manner.

Two other routes are set up in a similar manner with the `get()` function:

```
app.get('/collections',function(req,res){
  db.driver.collectionNames(function(e,names){
    res.json(names);
  })
});
app.get('/collections/:name',function(req,res){
  var collection = db.get(req.params.name);
  collection.find({},{limit:20},function(e,docs){
    res.json(docs);
  })
});
```

Express conveniently supports other HTTP verbs like post and update. In the case of setting up a post route, we write this:

```
app.post('product/:id',function(req,res) {...});
```

Express also has support for middleware. Middleware is just a request function handler with three parameters: `request`, `response`, and `next`. For example:

```
app.post('product/:id',
  authenticateUser,
  validateProduct,
  addProduct
);

function authenticateUser(req,res, next) {
  //check req.session for authentication
  next();
}

function validateProduct (req, res, next) {
   //validate submitted data
   next();
}

function addProduct (req, res) {
  //save data to database
}
```

validateProduct and authenticateProduct are middlewares. They are usually put into separate file (or files) in big projects.

Another way to set up middleware in the Express application is to utilize the `use()` function. For example, earlier we did this for static assets:

```
app.use(express.static(__dirname + '/public'));
```

We can also do it for error handlers:

```
app.use(errorHandler);
```

Assuming you have mongoDB installed, this app will connect to it (localhost:27017[51]) and display the collection name and items in collections. To start the mongo server:

```
$ mongod
```

to run the app (keep the **mongod** terminal window open):

[51]http://localhost:27017

```
$ node .
```

or

```
$ node index.js
```

To see the app working, open localhost:3000[52] in Chrome with the JSONViewer[53] extension (to render JSON nicely).

2.6 Intro to Express.js: Parameters, Error Handling and Other Middleware

2.6.1 Request Handlers

Express.js is a node.js framework that, among other things, provides a way to organize routes. Each route is defined via a method call on an application object with a URL patter as a first parameter (RegExp is also supported) — for example:

```
app.get('api/v1/stories/', function(res, req){
  ...
})
```

or, for a POST method:

```
app.post('/api/v1/stories'function(req,res){
  ...
})
```

Needless to say, DELETE and PUT methods are supported as well[54].

The callbacks that we pass to the `get()` or `post()` methods are called request handlers because they take requests (`req`), process them, and write to response (`res`) objects. For example:

```
app.get('/about', function(req,res){
  res.send('About Us: ...');
});
```

We can have multiple request handlers — hence the name *middleware*. They accept a third parameter, `next`, calling which (`next()`) will switch the execution flow to the next handler:

[52]http://localhost:3000
[53]https://chrome.google.com/webstore/detail/jsonview/chklaanhfefbnpoihckbnefhakgolnmc?hl=en
[54]http://expressjs.com/api.html#app.VERB

```
app.get('/api/v1/stories/:id', function(req,res, next) {
  //do authorization
  //if not authorized or there is an error
  // return next(error);
  //if authorized and no errors
  return next();
}), function(req,res, next) {
  //extract id and fetch the object from the database
  //assuming no errors, save story in the request object
  req.story = story;
  return next();
}), function(req,res) {
  //output the result of the database search
  res.send(res.story);
});
```

The ID of a story in URL patter is a query string parameter which we need for finding matching items in the database.

2.6.2 Parameters Middleware

Parameters are values passed in a query string of a URL of the request. If we didn't have Express.js or a similar library and had to use just the core Node.js modules, we'd have to extract parameters from HTTP.request[55] object via some `require('querystring').parse(url)` or `require('url').parse(url, true)` functions trickery.

Thanks to Connect framework[56] and the people at VisionMedia[57], Express.js already has support for parameters, error handling and many other important features in the form of middlewares. This is how we can plug param middleware in our app:

```
app.param('id', function(req,res, next, id){
  //do something with id
  //store id or other info in req object
  //call next when done
  next();
});

app.get('/api/v1/stories/:id',function(req,res){
  //param middleware will be execute before and
  //we expect req object already have needed info
  //output something
  res.send(data);
});
```

[55]http://nodejs.org/api/http.html#http_http_request_options_callback
[56]http://www.senchalabs.org/connect/
[57]https://github.com/visionmedia/express

For example:

```
app.param('id', function(req,res, next, id){
  req.db.get('stories').findOne({_id:id}, function (e, story){
    if (e) return next(e);
    if (!story) return next(new Error('Nothing is found'));
    req.story = story;
    next();
  });
});

app.get('/api/v1/stories/:id',function(req,res){
  res.send(req.story);
});
```

Or we can use multiple request handlers, but the concept remains the same: we can expect to have the `req.story` object or an error thrown prior to the execution of this code, so we abstract the common code/logic of getting parameters and their respective objects:

```
app.get('/api/v1/stories/:id', function(req,res, next) {
  //do authorization
  }),
  //we have an object in req.story so no work is needed here
  function(req,res) {
  //output the result of the database search
  res.send(story);
});
```

Authorization and input sanitation are also good candidates for residing in the middlewares.

The function `param()` is especially cool because we can combine different keys, e.g.:

```
app.get('/api/v1/stories/:storyId/elements/:elementId',
  function(req,res){
    res.send(req.element);
  }
);
```

2.6.3 Error Handling

Error handling is typically used across the whole application, so it's best to implement it as a middleware. It has the same parameters plus one more, `error`:

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.send(500);
})
```

In fact, the response can be anything:

JSON string

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.send(500, {status:500,
    message: 'internal error',
    type:'internal'}
  );
})
```

Text message

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.send(500, 'internal server error');
})
```

Error page

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  //assuming that template engine is plugged in
  res.render('500');
})
```

Redirect to error page

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.redirect('/public/500.html');
})
```

Error HTTP response status (401, 400, 500, etc.)

```
app.use(function(err, req, res, next) {
  //do logging and user-friendly error message display
  res.end(500);
})
```

By the way, logging should also be abstracted in a middleware!

To trigger an error from within your request handlers and middleware, you can just call:

```
next(error);
```

or

```
next(new Error('Something went wrong :-('));
```

You can also have multiple error handlers and use named instead of anonymous functions, as its shows in the Express.js Error handling guide[58].

2.6.4 Other Middleware

In addition to extracting parameters, it can be used for many things, like authorization, error handling, sessions, output, and others.

`res.json()` is one of them. It conveniently outputs the JavaScript/Node.js object as a JSON. For example:

```
app.get('/api/v1/stories/:id', function(req,res){
  res.json(req.story);
});
```

is equivalent to (if `req.story` is an Array and Object):

```
app.get('/api/v1/stories/:id', function(req,res){
  res.send(req.story);
});
```

or

[58]http://expressjs.com/guide.html#error-handling

```
app.get('api/v1/stories/:id',function(req,res){
  res.set({
    'Content-Type': 'application/json'
  });
  res.send(req.story);
});
```

2.6.5 Abstraction

Middleware is flexible. You can use anonymous or named functions, but the best thing is to abstract request handlers into external modules based on the functionality:

```
var stories = require.('./routes/stories');
var elements = require.('./routes/elements');
var users = require.('./routes/users');
...
app.get('/stories/,stories.find);
app.get('/stories/:storyId/elements/:elementId', elements.find);
app.put('/users/:userId',users.update);
```

routes/stories.js:

```
module.exports.find = function(req,res, next) {
};
```

routes/elements.js:

```
module.exports.find = function(req,res,next){
};
```

routes/users.js:

```
module.exports.update = function(req,res,next){
};
```

You can use some functional programming tricks, like this:

```
function requiredParamHandler(param){
  //do something with a param, e.g.,
  //check that it's present in a query string
  return function (req,res, next) {
    //use param, e.g., if token is valid proceed with next();
    next();
  });
}

app.get('/api/v1/stories/:id',
  requiredParamHandler('token'),
  story.show
);

var story  = {
  show: function (req, res, next) {
    //do some logic, e.g., restrict fields to output
    return res.send();
  }
}
```

As you can see, middleware is a powerful concept for keeping code organized. The best practice is to keep the router lean and thin by moving all of the logic into corresponding external modules/files. This way, important server configuration parameters will be neatly in one place when you need them! :-)

2.7 JSON REST API server with Node.js and MongoDB using Mongoskin and Express.js

This tutorial will walk you through writing test using the Mocha[59] and Super Agent[60] libraries and then use them in a test-driven development manner to build a Node.js[61] free JSON REST API server utilizing Express.js[62] framework and Mongoskin[63] library for MongoDB[64]. In this REST API server, we'll perform **create, update, remove and delete** (CRUD) operations and harness Express.js middleware[65] concept with `app.param()` and `app.use()` methods.

[59]http://visionmedia.github.io/mocha/
[60]http://visionmedia.github.io/superagent/
[61]http://nodejs.org
[62]http://expressjs.com/
[63]https://github.com/kissjs/node-mongoskin
[64]http://www.mongodb.org/
[65]http://expressjs.com/api.html#middleware

2.7.1 Test Coverage

Before anything else let's write functional tests that make HTTP requests to our soon-to-be-created REST API server. If you know how to use Mocha[66] or just want to jump straight to the Express.js app implementation feel free to do so. You can use CURL terminal commands for testing too.

Assuming we already have Node.js, NPM[67] and MongoDB installed, let's create a *new* folder (or if you wrote the tests use that folder):

```
mkdir rest-api
cd rest-api
```

We'll use Mocha[68], Expect.js[69] and Super Agent[70] libraries. To install them run these command from the project folder:

```
$ npm install mocha
$ npm install expect.js
$ npm install superagent
```

Now let's create **express.test.js** file in the same folder which will have six suites:

- creating a new object
- retrieving an object by its ID
- retrieving the whole collection
- updating an object by its ID
- checking an updated object by its ID
- removing an object by its ID

HTTP requests are just a breeze with Super Agent's chained functions which we'll put inside of each test suite. Here is the full source code for the **express.test.js** file:

[66] http://visionmedia.github.io/mocha/
[67] http://npmjs.org
[68] http://visionmedia.github.io/mocha/
[69] https://github.com/LearnBoost/expect.js/
[70] http://visionmedia.github.io/superagent/

```
var superagent = require('superagent')
var expect = require('expect.js')

describe('express rest api server', function(){
  var id

  it('post object', function(done){
    superagent.post('http://localhost:3000/collections/test')
      .send({ name: 'John'
        , email: 'john@rpjs.co'
      })
      .end(function(e,res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(res.body.length).to.eql(1)
        expect(res.body[0]._id.length).to.eql(24)
        id = res.body[0]._id
        done()
      })
  })

  it('retrieves an object', function(done){
    superagent.get('http://localhost:3000/collections/test/'+id)
      .end(function(e, res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(typeof res.body).to.eql('object')
        expect(res.body._id.length).to.eql(24)
        expect(res.body._id).to.eql(id)
        done()
      })
  })

  it('retrieves a collection', function(done){
    superagent.get('http://localhost:3000/collections/test')
      .end(function(e, res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(res.body.length).to.be.above(1)
        expect(res.body.map(function (item){
          return item._id
        })).to.contain(id)
        done()
      })
  })

  it('updates an object', function(done){
    superagent.put('http://localhost:3000/collections/test/'+id)
      .send({name: 'Peter'
        , email: 'peter@yahoo.com'})
      .end(function(e, res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(typeof res.body).to.eql('object')
        expect(res.body.msg).to.eql('success')
        done()
      })
  })

  it('checks an updated object', function(done){
    superagent.get('http://localhost:3000/collections/test/'+id)
      .end(function(e, res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(typeof res.body).to.eql('object')
        expect(res.body._id.length).to.eql(24)
        expect(res.body._id).to.eql(id)
        expect(res.body.name).to.eql('Peter')
        done()
      })
  })

  it('removes an object', function(done){
    superagent.del('http://localhost:3000/collections/test/'+id)
      .end(function(e, res){
        // console.log(res.body)
        expect(e).to.eql(null)
        expect(typeof res.body).to.eql('object')
        expect(res.body.msg).to.eql('success')
        done()
      })
  })
})
```

To run the tests we can use the `$ mocha express.test.js` command.

2.7.2 Dependencies

In this tutorial we'll utilize Mongoskin[71], a MongoDB library which is a better alternative to the plain good old native MongoDB driver for Node.js[72]. In additition Mongoskin is more light-weight than Mongoose and schema-less. For more insight please check out Mongoskin comparison blurb[73].

Express.js[74] is a wrapper for the core Node.js HTTP module[75] objects. The Express.js framework is build on top of Connect[76] middleware and provided tons of convenience. Some people compare the framework to Ruby's Sinatra in terms of how it's non-opinionated and configurable.

If you've create a `rest-api` folder in the previous section *Test Coverage*, simply run these commands to install modules for the application:

```
npm install express
npm install mongoskin
```

2.7.3 Implementation

First things first, so let's define our dependencies:

```
var express = require('express')
  , mongoskin = require('mongoskin')
```

After the version 3.x, Express streamlines the instantiation of its app instance, in a way that this line will give us a server object:

```
var app = express()
```

To extract params from the body of the requests we'll use `bodyParser()` middleware which looks more like a configuration statement:

```
app.use(express.bodyParser())
```

Middleware (in this[77] and other forms[78]) is a powerful and convenient pattern in Express.js and Connect[79] to organize and re-use code.

As with the `bodyParser()` method that saves us from the hurdles of parsing a body object of HTTP request, Mongoskin makes possible to connect to the MongoDB database in one effortless line of code:

[71]https://github.com/kissjs/node-mongoskin
[72]https://github.com/mongodb/node-mongodb-native
[73]https://github.com/kissjs/node-mongoskin#comparation
[74]http://expressjs.com/
[75]http://nodejs.org/api/http.html
[76]https://github.com/senchalabs/connect
[77]http://expressjs.com/api.html#app.use
[78]http://expressjs.com/api.html#middleware
[79]https://github.com/senchalabs/connect

```
var db = mongoskin.db('localhost:27017/test', {safe:true});
```

Note: *If you wish to connect to a remote database, e.g., MongoHQ[80] instance, substitute the string with your username, password, host and port values. Here is the format of the URI string:*`mongodb://[username:password@]host1[:port1`

The `app.param()` method is another Express.js middleware. It basically says "*do something every time there is this value in the URL pattern of the request handler.*" In our case we select a particular collection when request pattern contains a sting `collectionName` prefixed with a colon (you'll see it later in the routes):

```
app.param('collectionName',
  function(req, res, next, collectionName) {
    req.collection = db.collection(collectionName)
    return next()
  }
)
```

Merely to be user-friendly, let's put a root route with a message:

```
app.get('/', function(req, res) {
  res.send('please select a collection, e.g., /collections/messages')
})
```

Now the real work begins. Here is how we retrieve a list of items sorted by `_id` and which has a limit of 10:

```
app.get('/collections/:collectionName',
  function(req, res) {
    req.collection
      .find({},
        {limit:10, sort: [['_id',-1]]}
      ).toArray(function(e, results){
        if (e) return next(e)
        res.send(results)
      }
    )
  }
)
```

Have you noticed a `:collectionName` string in the URL pattern parameter? This and the previous `app.param()` middleware is what gives us the `req.collection` object which points to a specified collection in our database.

The object creating endpoint is slightly easier to grasp since we just pass the whole payload to the MongoDB (method a.k.a. free JSON REST API):

[80]https://www.mongohq.com/home

```
app.post('/collections/:collectionName', function(req, res) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
```

Single object retrieval functions are faster than `find()`, but they use different interface (they return object directly instead of a cursor), so please be aware of that. In addition, we're extracting the ID from `:id` part of the path with `req.params.id` Express.js magic:

```
app.get('/collections/:collectionName/:id', function(req, res) {
  req.collection.findOne({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send(result)
    }
  )
})
```

PUT request handler gets more interesting because `update()` doesn't return the augmented object, instead it returns us a count of affected objects.

Also `{$set:req.body}` is a special MongoDB operator (operators tend to start with a dollar sign) that sets values.

The second ‘ {safe:true, multi:false}‘ parameter is an object with options that tell MongoDB to wait for the execution before running the callback function and to process only one (first) item.

```
app.put('/collections/:collectionName/:id', function(req, res) {
  req.collection.update({_id: req.collection.id(req.params.id)},
    {$set:req.body},
    {safe:true, multi:false},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  )
})
```

Finally, the DELETE method which also output a custom JSON message:

```
app.del('/collections/:collectionName/:id', function(req, res) {
  req.collection.remove({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  )
})
```

Note: *The* `delete` *is an operator in JavaScript, so Express.js uses* `app.del` *instead.*

The last line that actually starts the server on port 3000 in this case:

```
app.listen(3000)
```

Just in case something is not working quite well here is the full code of **express.js** file:

```
var express = require('express')
  , mongoskin = require('mongoskin')

var app = express()
app.use(express.bodyParser())

var db = mongoskin.db('localhost:27017/test', {safe:true});

app.param('collectionName',
  function(req, res, next, collectionName){
    req.collection = db.collection(collectionName)
    return next()
  }
)
app.get('/', function(req, res) {
  res.send('please select a collection, '
    + 'e.g., /collections/messages')
})
app.get('/collections/:collectionName', function(req, res) {
  req.collection.find({},{limit:10, sort: [['_id',-1]]})
    .toArray(function(e, results){
      if (e) return next(e)
      res.send(results)
    }
  )
})

app.post('/collections/:collectionName', function(req, res) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})
app.get('/collections/:collectionName/:id', function(req, res) {
  req.collection.findOne({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send(result)
    }
  )
})
app.put('/collections/:collectionName/:id', function(req, res) {
  req.collection.update({_id: req.collection.id(req.params.id)},
    {$set:req.body},
    {safe:true, multi:false},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  )
})
app.del('/collections/:collectionName/:id', function(req, res) {
  req.collection.remove({_id: req.collection.id(req.params.id)},
    function(e, result){
      if (e) return next(e)
      res.send((result===1)?{msg:'success'}:{msg:'error'})
    }
  )
})

app.listen(3000)
```

Exit your editor and run this in your terminal:

```
$ node express.js
```

And in a different window (without closing the first one):

```
$ mocha express.test.js
```

If you really don't like Mocha and/or BDD, CURL is always there for you. :-)

For example, CURL data to make a POST request:

```
$ curl -d "" http://localhost:3000
```

GET requests also work in the browser, for example http://localhost:3000/test.

In this tutorial our tests are longer than the app code itself so abandoning test-driven development might be tempting, but believe me **the good habits of TDD will save you hours and hours** during any serious development when the complexity of the applications you work one is big.

2.7.4 Conclusion

The Express.js and Mongoskin libraries are great when you need to build a simple REST API server in a few line of code. Later, if you need to expand the libraries they also provide a way to configure and organize your code.

NoSQL databases like MongoDB are good at free-REST APIs where we don't have to define schemas and can throw any data and it'll be saved.

The full code of both test and app files: https://gist.github.com/azat-co/6075685.

If you like to learn more about Express.js and other JavaScript libraries take a look at the series Intro to Express.js tutorials[81].

Note: *In this example I'm using semi-colon less style. Semi-colons in JavaScript are absolutely optional[82] except in two cases: in the for loop and before expression/statement that starts with parenthesis (e.g., Immediately-Invoked Function Expression[83]).

2.8 Node.js MVC: Express.js + Derby Hello World Tutorial

2.8.1 Node MVC Framework

Express.js[84] is a popular node framework which uses the middleware concept to enhance the functionality of applications. Derby[85] is a new sophisticated Model View Controller (MVC[86]) framework which is designed to be used with Express[87] as its middleware. Derby also comes with the support of Racer[88], data synchronization engine, and a Handlebars[89]-like template engine, among many other features[90].

[81]http://webapplog.com/tag/intro-to-express-js/
[82]http://blog.izs.me/post/2353458699/an-open-letter-to-javascript-leaders-regarding
[83]http://en.wikipedia.org/wiki/Immediately-invoked_function_expression
[84]http://expressjs.com
[85]http://derbyjs.com
[86]http://en.wikipedia.org/wiki/Model%E2%80%93view%E2%80%93controller
[87]http://expressjs.com
[88]https://github.com/codeparty/racer
[89]https://github.com/wycats/handlebars.js/
[90]http://derbyjs.com/#features

2.8.2 Derby Installation

Let's set up a basic Derby application architecture without the use of scaffolding. Usually project generators are confusing when people just start to learn a new comprehensive framework. This is a bare minimum "Hello World" application tutorial that still illustrates the Derby skeleton and demonstrates live-templates with websockets.

Of course, we'll need Node.js[91] and NPM[92], which can be obtained at nodejs.org[93]. To install Derby globally, run:

```
$ npm install -g derby
```

To check the installation:

```
$ derby -V
```

My version, as of April 2013, is 0.3.15. We should be good to go to creating our first app!

2.8.3 File Structure

This is the project folder structure:

```
project/
  -package.json
  -index.js
  -derby-app.js
  views/
    derby-app.html
  styles/
    derby-app.less
```

2.8.4 Dependencies

Let's include dependencies and other basic information in the **package.json** file:

[91]http://nodejs.org
[92]http://npmjs.org
[93]http://nodejs.org

```
 {
  "name": "DerbyTutorial",
  "description": "",
  "version": "0.0.0",
  "main": "./server.js",
  "dependencies": {
    "derby": "*",
    "express": "3.x"
  },
  "private": true
}
```

Now we can run `npm install`, which will download our dependencies into **node_modules** folder.

2.8.5 Views

Views must be in the **views** folder, and they must be either in **index.html** under a folder which has the same name as your derby app JavaScript file, i.e., `views/derby-app/index.html`, or be inside of a file which has the same name as your derby app JS file, i.e., **derby-app.html**.

In this example, the "Hello World" app, we'll use `<Body:>` template and `{message}` variable. Derby uses mustach[94]-handlebars-like syntax for reactive binding. **index.html** looks like this:

```
<Body:>
  <input value="{message}"><h1>{message}</h1>
```

Same thing with Stylus/LESS files; in our example, index.css has just one line:

```
h1 {
  color: blue;
}
```

To find out more about those wonderful CSS preprocessors, check out the documentation at Stylus[95] and LESS[96].

2.8.6 Main Server

index.js is our main server file, and we begin it with an inclusion of dependencies with the `require()` function:

[94]http://mustache.github.io/
[95]http://learnboost.github.io/stylus/
[96]http://lesscss.org/

```
var http = require('http'),
  express = require('express'),
  derby = require('derby'),
  derbyApp = require('./derby-app');
```

The last line is our derby application file **derby-app.js**.

Now we're creating the Express.js application (v3.x has significant differences between 2.x) and an HTTP server:

```
var expressApp = new express(),
  server = http.createServer(expressApp);
```

Derby[97] uses the Racer[98] data synchronization library, which we create like this:

```
var store = derby.createStore({
  listen: server
});
```

To fetch some data from back-end to the front-end, we instantiate the model object:

```
var model = store.createModel();
```

Most importantly we need to pass the model and routes as middlewares to the Express.js app. We need to expose the public folder for socket.io to work properly.

```
expressApp.
  use(store.modelMiddleware()).
  use(express.static(__dirname + '/public')).
  use(derbyApp.router()).
  use(expressApp.router);
```

Now we can start the server on port 3001 (or any other):

```
server.listen(3001, function(){
  model.set('message', 'Hello World!');
});
```

Full code of **index.js** file:

[97]http://derbyjs.com
[98]https://github.com/codeparty/racer

```
var http = require('http'),
  express = require('express'),
  derby = require('derby'),
  derbyApp = require('./derby-app');

var expressApp = new express(),
  server = http.createServer(expressApp);

var store = derby.createStore({
  listen: server
});

var model = store.createModel();

expressApp.
  use(store.modelMiddleware()).
  use(express.static(__dirname + '/public')).
  use(derbyApp.router()).
  use(expressApp.router);

server.listen(3001, function(){
  model.set('message', 'Hello World!');
});
```

2.8.7 Derby Application

Finally, the Derby app file, which contains code for both a front-end and a back-end. Front-end only code is inside of the `app.ready()` callback. To start, let's require and create an app. Derby uses unusual construction (not the same familiar good old `module.exports = app`):

```
var derby = require('derby'),
  app = derby.createApp(module);
```

To make socket.io magic work, we need to subscribe a model attribute to its visual representation — in other words, bind data and view. We can do it in the root route, and this is how we define it (patter is `/`, a.k.a. root):

```
app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  });
});
```

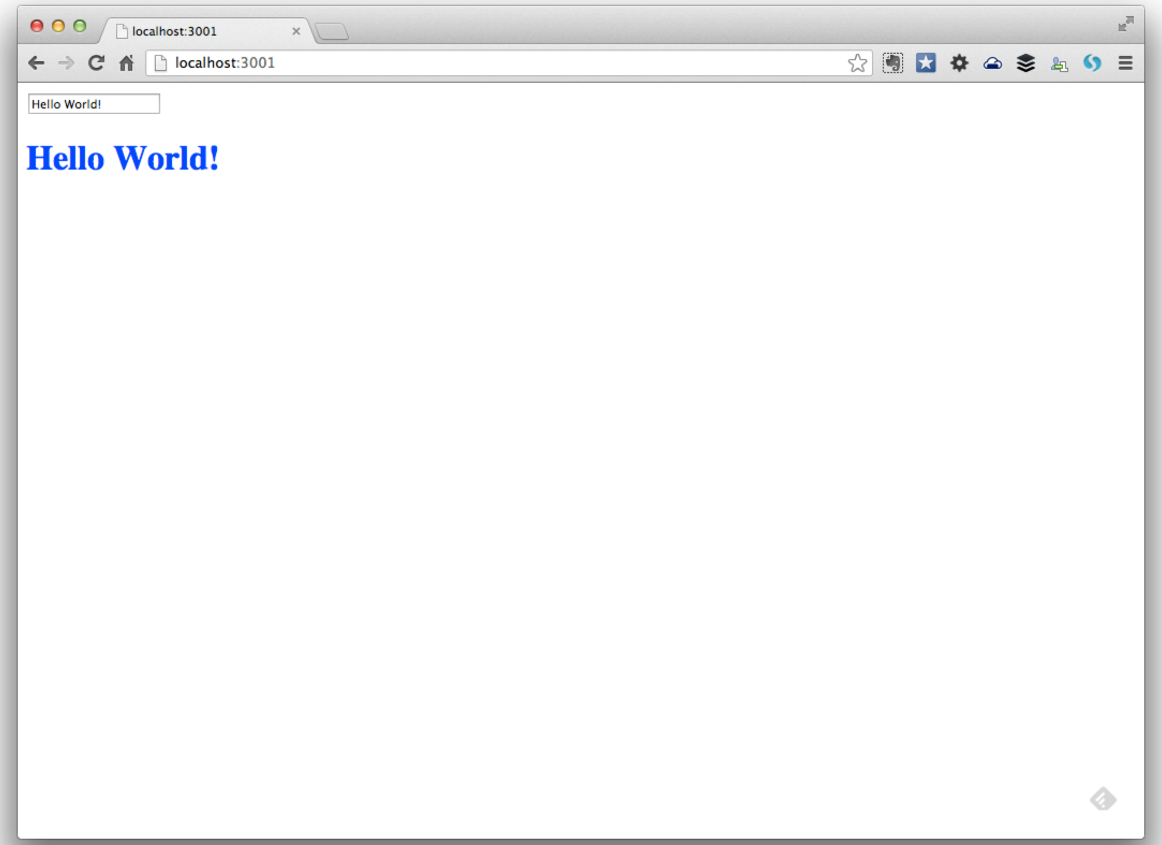
Full code of **derby-app.js** file:

```
var derby = require('derby'),
  app = derby.createApp(module);

app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  });
});
```

2.8.8 Launching Hello World App

Now everything should be ready to boot our server. Execute `node .` or `node index.js` and open a browser at localhost:3001[99]. You should be able to see something like this:



Hello World!

Derby + Express.js Hello World App

[99]http://localhost:3001

2.8.9 Passing Values to Back-End

Of course, the static data is not much, so we can slightly modify our app to make back-end and front-end pieces talks with each other.

In the server file **index.js**, add `store.afterDb` to listen to `set` events on the **message** attribute:

```
server.listen(3001, function(){
  model.set('message', 'Hello World!');
  store.afterDb('set','message', function(txn, doc, prevDoc, done){
    console.log(txn)
    done();
  });
});
```

Full code of **index.js** after modifications:

```
var http = require('http'),
  express = require('express'),
  derby = require('derby'),
  derbyApp = require('./derby-app');

var expressApp = new express(),
  server = http.createServer(expressApp);

var store = derby.createStore({
  listen: server
});

var model = store.createModel();

expressApp.
  use(store.modelMiddleware()).
  use(express.static(__dirname + '/public')).
  use(derbyApp.router()).
  use(expressApp.router);

server.listen(3001, function(){
  model.set('message', 'Hello World!');
  store.afterDb('set','message', function(txn, doc, prevDoc, done){
    console.log(txn)
    done();
  });
});
```

In the Derby application file **derby-app.js**, add `model.on()` to `app.ready()`:

```
  app.ready(function(model){
            model.on('set', 'message',function(path, object){
            console.log('message has been changed: '+ object);
          })
  });
```

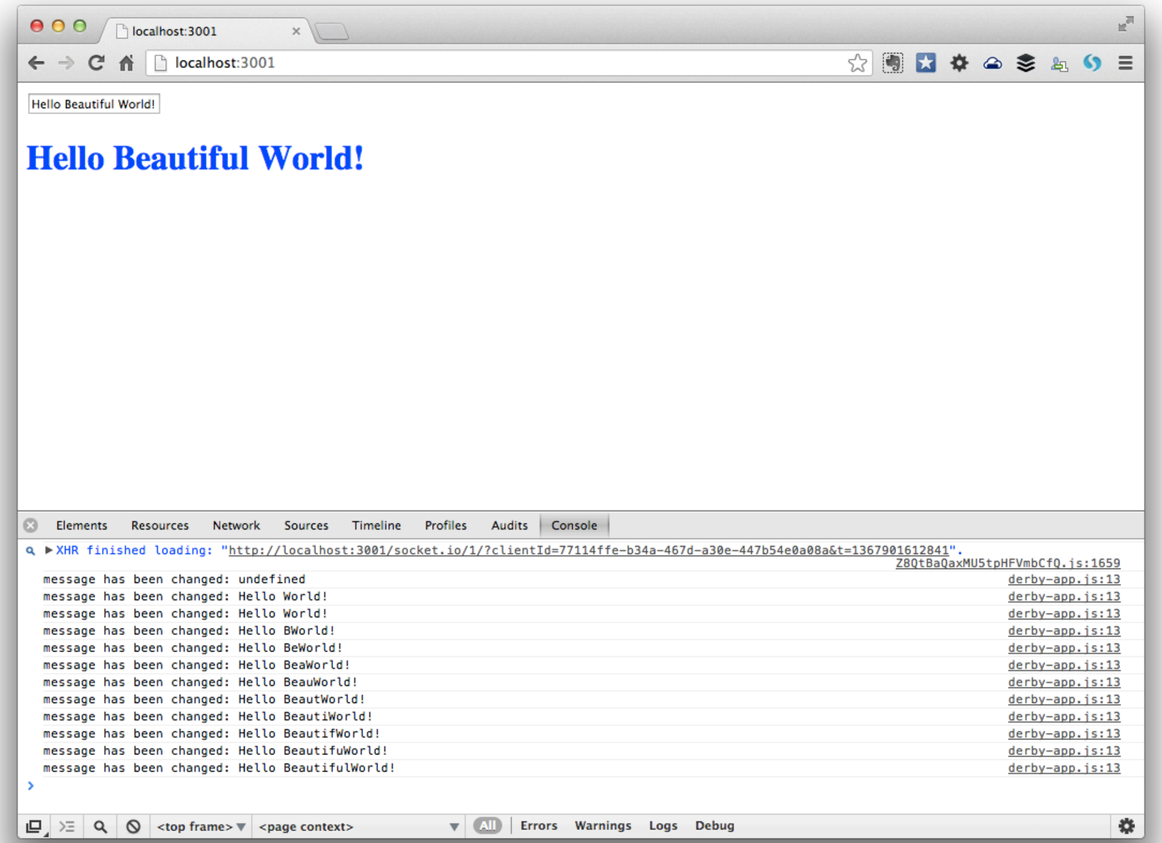
Full **derby-app.js** file after modifications:

```
var derby = require('derby'),
  app = derby.createApp(module);

app.get('/', function(page, model, params) {
  model.subscribe('message', function() {
    page.render();
  })
});

app.ready(function(model) {
  model.on('set', 'message', function(path, object) {
    console.log('message has been changed: ' + object);
  })
});
```

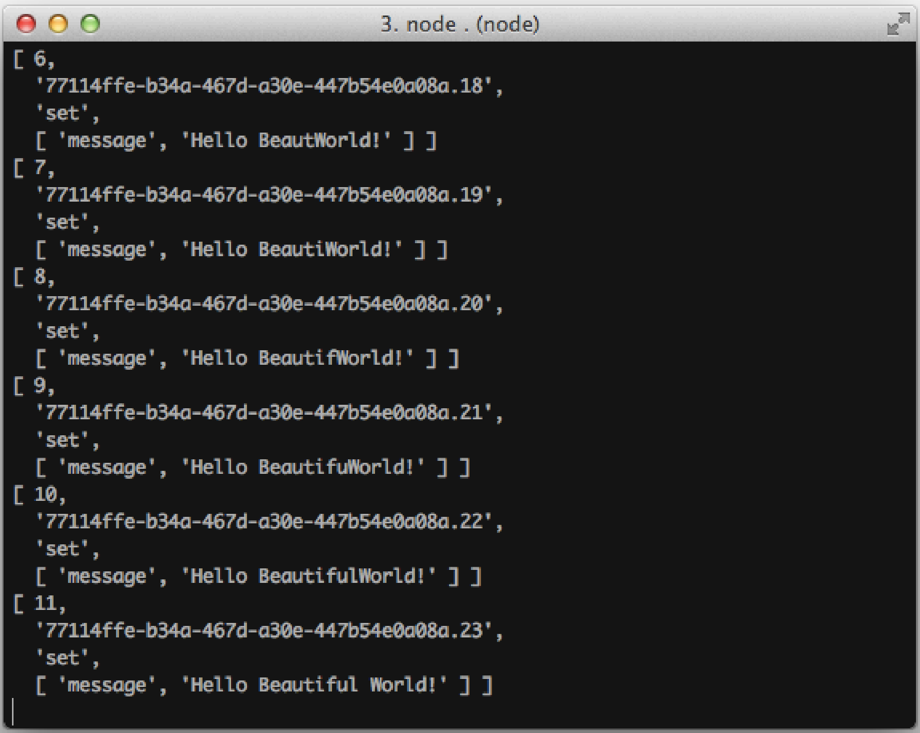
Now we'll see logs both in the terminal window and in the browser Developer Tools console. The end result should look like this in the browser:



```
▶XHR finished loading: "http://localhost:3001/socket.io/1/?clientId=77114ffe-b34a-467d-a30e-447b54e0a08a&t=1367901612841".
                                                                                     Z8QtBaQaxMU5tpHFVmbCfQ.js:1659
message has been changed: undefined                                                               derby-app.js:13
message has been changed: Hello World!                                                            derby-app.js:13
message has been changed: Hello World!                                                            derby-app.js:13
message has been changed: Hello BWorld!                                                           derby-app.js:13
message has been changed: Hello BeWorld!                                                          derby-app.js:13
message has been changed: Hello BeaWorld!                                                         derby-app.js:13
message has been changed: Hello BeauWorld!                                                        derby-app.js:13
message has been changed: Hello BeautWorld!                                                       derby-app.js:13
message has been changed: Hello BeautiWorld!                                                      derby-app.js:13
message has been changed: Hello BeautifWorld!                                                     derby-app.js:13
message has been changed: Hello BeautifuWorld!                                                    derby-app.js:13
message has been changed: Hello BeautifulWorld!                                                   derby-app.js:13
>
```

Hello World App: Browser Console Logs

And like this in the terminal:



```
[ 6,
  '77114ffe-b34a-467d-a30e-447b54e0a08a.18',
  'set',
  [ 'message', 'Hello BeautWorld!' ] ]
[ 7,
  '77114ffe-b34a-467d-a30e-447b54e0a08a.19',
  'set',
  [ 'message', 'Hello BeautiWorld!' ] ]
[ 8,
  '77114ffe-b34a-467d-a30e-447b54e0a08a.20',
  'set',
  [ 'message', 'Hello BeautifWorld!' ] ]
[ 9,
  '77114ffe-b34a-467d-a30e-447b54e0a08a.21',
  'set',
  [ 'message', 'Hello BeautifuWorld!' ] ]
[ 10,
  '77114ffe-b34a-467d-a30e-447b54e0a08a.22',
  'set',
  [ 'message', 'Hello BeautifulWorld!' ] ]
[ 11,
  '77114ffe-b34a-467d-a30e-447b54e0a08a.23',
  'set',
  [ 'message', 'Hello Beautiful World!' ] ]
```

Hello World App: Terminal Console Logs

For more magic in the persistence area, check out Racer's db property[100]. With it you can set up an automatic synch between views and database!

The full code of all of the files in this Express.js + Derby Hello World app is available as a gist at gist.github.com/azat-co/5530311[101].

[100]http://derbyjs.com/#persistence
[101]https://gist.github.com/azat-co/5530311

3. Introduction

Summary: reasons behind rapid prototyping in general and writing of this book; answers to questions what to expect and what not, what are prerequisites; suggestions on how to use the book and examples; explanation of book's notation format.

"Get out of the building." — Steve Blank[1]

Rapid Prototyping with JS is a hands-on book which introduces you to rapid software prototyping using the latest cutting-edge web and mobile technologies including Node.js[2], MongoDB[3], Twitter Bootstrap[4], LESS[5], jQuery[6], Parse.com[7], Heroku[8] and others.

3.1 Why RPJS?

This book was borne out of frustration. I have been in software engineering for many years, and when I started learning Node.js and Backbone.js, I learned the hard way that their official documentation and the Internet lack in quick start guides and examples. Needless to say, it was virtually impossible to find all of the tutorials for JS-related modern technologies in one place.

The best way to *learn* is to *do*, right? Therefore, I've used the approach of small simple examples, i.e., quick start guides, to expose myself to the new cool tech. After I was done with the basic apps, I needed some references and organization. I started to write this manual mostly for myself, so I can understand the concepts better and refer to the samples later. Then StartupMonthly[9] and I taught a few 2-day intensive classes on the same subject — helping experienced developers to jump-start their careers with agile JavaScript development. The manual we used was updated and iterated many times based on the feedback received. The end result is this book.

3.2 What to Expect

A typical reader of RPJS should expect a collection of quick start guides, tutorials and suggestions (e.g., Git workflow). There is a lot of coding and not much theory. All the theory we cover is directly related to some of

[1]http://steveblank.com/
[2]http://nodejs.org
[3]http://mongodb.org
[4]http://twitter.github.com/bootstrap
[5]http://lesscss.org
[6]http://jquery.com
[7]http://parse.com
[8]http://heroku.com
[9]http://startupmonthly.org

the practical aspects, and essential for better understanding of technologies and specific approaches in dealing with them, e.g., JSONP and cross-domain calls.

In addition to coding examples, the book covers virtually all setup and deployment step-by-step.

You'll learn on the examples of Chat web/mobile applications starting with front-end components. There are a few versions of these applications, but by the end we'll put front-end and back-end together and deploy to the production environment. The Chat application contains all of the necessary components typical for a basic web app, and will give you enough confidence to continue developing on your own, apply for a job/promotion or build a startup!

3.3 Who This Book is For

The book is designed for advanced-beginner and intermediate-level web and mobile developers: somebody who has been (or still is) an expert in other languages like Ruby on Rails, PHP, Perl, Python or/and Java. The type of a developer who wants to learn more about JavaScript and Node.js related techniques for building web and mobile application prototypes *fast*. Our target user doesn't have time to dig through voluminous (or tiny, at the other extreme) official documentation. The goal of *Rapid Prototyping with JS* is not to make an expert out of a reader, but to help him/her to start building apps as soon as possible.

Rapid Prototyping with JS: Agile JavaScript Development, as you can tell from the name, is about taking your idea to a functional prototype in the form of a web or a mobile application as fast as possible. This thinking adheres to the Lean Startup[10] methodology; therefore, this book would be more valuable to startup founders, but big companies' employees might also find it useful, especially if they plan to add new skills to their resumes.

3.4 What This Book is Not

Rapid Prototyping with JS is **neither** a comprehensive book on several frameworks, libraries or technologies (or just a particular one), **nor** a reference for all the tips and tricks of web development. Examples similar to ones in this book might be *publicly* available online.

Even more so, if you're not familiar with fundamental programming concepts like loops, if/else statements, arrays, hashes, object and functions, you won't find them in *Rapid Prototyping with JS*. Additionally, it would be challenging to follow our examples.

Many volumes of great books have been written on fundamental topics — the list of such resources is at the end of the book in the chapter *Further Reading*. The purpose of *Rapid Prototyping with JS* is to give agile tools without replicating theory of programming and computer science.

3.5 Prerequisites

We recommend the following things to get the full advantage of the examples and materials covered:

[10]http://theleanstartup.com

- Knowledge of the fundamental programming concepts such as objects, functions, data structures (arrays, hashes), loops (for, while), conditions (if/else, switch)
- Basic web development skills including, but not limited to, HTML and CSS
- Mac OS X or UNIX/Linux systems are highly recommended for this book's examples and for web development in general, although it's still possible to hack your way on a Windows-based system
- Access to the Internet
- 5-20 hours of time
- Some cloud services require users' credit/debit card information even for free accounts

3.6 How to Use the Book

For soft-copy (digital version) the book comes in three formats:

1. PDF: suited for printing; opens in Adobe Reader, Mac OS X Preview, iOS apps, and other PDF viewers.
2. ePub: suited for iBook app on iPad and other iOS devices; to copy to devices use iTunes, Dropbox or email to yourself.
3. mobi: suited for Kindles of all generations as well as desktop and mobile Amazon Kindle apps and Amazon Cloud Reader; to copy to devices use Whispernet, USB cable or email to yourself.

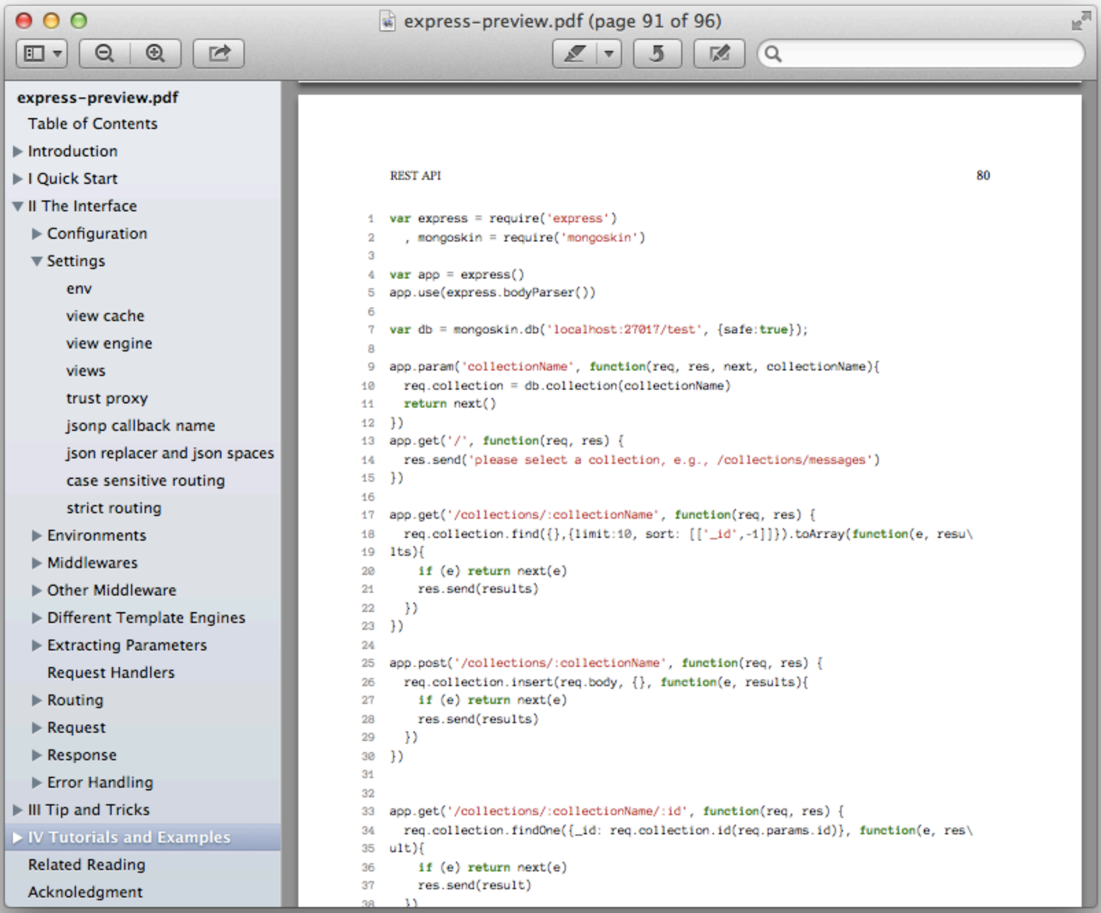
This is a digital version of the book, so most of the links are hidden just like on any other web page, e.g., jQuery[11] instead of http://jquery.com. In the PDF version, URLs are in the footnotes at the bottom of the page. The table of contents has local hyperlinks which allow you to jump to any part or chapter of the book.

There are summaries in the beginning of each chapter describing in a few short sentences what examples and topics the particular chapter covers.

In PDF, EPUB and Mobi versions you could use the **Table of Contents**, which is in the beginning of the book and has internal links, to jump to the most interesting parts or chapters.

For faster navigation between parts, chapters and sections of the book, please use book's navigation pane which is based on the **Table of Contents** (the screenshot is below).

[11]http://jquery.com



express-preview.pdf (page 91 of 96)

REST API 80

```
var express = require('express')
  , mongoskin = require('mongoskin')

var app = express()
app.use(express.bodyParser())

var db = mongoskin.db('localhost:27017/test', {safe:true});

app.param('collectionName', function(req, res, next, collectionName){
  req.collection = db.collection(collectionName)
  return next()
})
app.get('/', function(req, res) {
  res.send('please select a collection, e.g., /collections/messages')
})

app.get('/collections/:collectionName', function(req, res) {
  req.collection.find({},{limit:10, sort: [['_id',-1]]}).toArray(function(e, resu\
lts){
    if (e) return next(e)
    res.send(results)
  })
})

app.post('/collections/:collectionName', function(req, res) {
  req.collection.insert(req.body, {}, function(e, results){
    if (e) return next(e)
    res.send(results)
  })
})


app.get('/collections/:collectionName/:id', function(req, res) {
  req.collection.findOne({_id: req.collection.id(req.params.id)}, function(e, res\
ult){
    if (e) return next(e)
    res.send(result)
  })
```

The Table of Contents pane in the Mac OS X Preview app.

3.7 Examples

All of the source code for examples used in this book is available in the book itself for the most part, as well as in a public GitHub repository github.com/azat-co/rpjs[12]. You can also download files as a ZIP archive[13] or use Git to pull them. More on how to install and use Git will be covered later in the book. The source code files, folder structure and deployment files are supposed to work locally and/or remotely on PaaS solutions, i.e., Windows Azure and Heroku, with minor or no modifications.

Source code which is in the book is technically limited by the platform to the width of about 70 characters. We tried our best to preserve the best JavaScript and HTML formatting styles, but from time to time you might see backslashes (\). There is nothing wrong with the code. Backslashes are line escape characters, and if you

[12]http://github.com/azat-co/rpjs

[13]https://github.com/azat-co/rpjs/archive/master.zip

copy-paste the code into the editor, the example should work just fine. Please note that code in GitHub and in the book might differ in formatting. Also, let us know via email (hi@rpjs.co[14]) if you spot any bugs!

3.8 Notation

This is what source code blocks look like:

```
var object = {};
object.name = "Bob";
```

Terminal commands have a similar look but start with dollar sign or $:

```
$ git push origin heroku
$ cd /etc/
$ ls
```

Inline file names, path/folder names, quotes and special words/names are *italicized*, while command names, e.g., **mongod**, and emphasized words, e.g., **Note**, are **bold**.

3.9 Terms

For the purpose of this book, we're using some terms interchangeably, while depending on the context, they might not mean exactly the same thing. For example, function = method = call, attribute = property = member = key, value = variable, object = hash = class, list = array, framework = library = module.

[14]mailto:hi@rpjs.co

4. What Readers Say

> "Azat's tutorials are crucial to the development of Sidepon.com[1] interactive UX and the success of getting us featured on TheNextWeb.com[2] and reached profitability." — Kenson Goo (Sidepon.com[3])

> "I had a lot of fun reading this book and following its examples! It showcases and helps you discover a huge variety of technologies that everyone should consider using in their own projects." — Chema Balsas

Rapid Prototyping with JS is being successfully used at StartupMonthly[4] as a training[5] manual. Here are some of our trainees' testimonials:

> "Thanks a lot to all and special thanks to Azat and Yuri. I enjoyed it a lot and felt motivated to work hard to know these technologies." — Shelly Arora

> "Thanks for putting this workshop together this weekend… what we did with Bootstrap + Parse was really quick & awesome." — Mariya Yao

> "Thanks Yuri and all of you folks. It was a great session - very educative, and it certainly helped me brush up on my Javascript skills. Look forward to seeing/working with you in the future." — Sam Sur

[1]http://Sidepon.com
[2]http://thenextweb.com
[3]http://Sidepon.com
[4]http://startupmonthly.org
[5]http://www.startupmonthly.org/rapid-prototyping-with-javascript-and-nodejs.html

5. Rapid Prototyping with JS on the Internet

Let's be Friends on the Internet

- Twitter: @RPJSbook[1] and @azat_co[2]
- Facebook: facebook.com/RapidPrototypingWithJS[3]
- Website: rapidprototypingwithjs.com[4]
- Blog: webapplog.com[5]
- GitHub: github.com/azat-co/rpjs[6]
- Storify: Rapid Prototyping with JS[7]

Other Ways to Reach Us

- Email: hi@rpjs.co[8]
- Google Group: rpjs@googlegroups.com[9] and https://groups.google.com/forum/#!forum/rpjs

Share on Twitter

> *"I'm reading Rapid Prototyping with JS — book on agile development with JavaScript and Node.js by @azat_co #RPJS @RPJSbook"* — http://clicktotweet.com/biWsd

[1]https://twitter.com/rpjsbook
[2]https://twitter.com/azat_co
[3]https://www.facebook.com/RapidPrototypingWithJS
[4]http://rapidprototypingwithjs.com/
[5]http://webapplog.com
[6]https://github.com/azat-co/rpjs
[7]https://storify.com/azat_co/rapid-prototyping-with-js
[8]mailto:hi@rpjs.co
[9]mailto:rpjs@googlegroups.com

6. About the Author

Azat Mardan: a software engineer, an author and a yogi.

Azat Mardan has over 12 years of experience in web, mobile and software development. With a Bachelor's Degree in Informatics and a Master of Science in Information Systems Technology degree, Azat possesses deep academic knowledge as well as extensive practical experience.

Currently, Azat works as a Senior Software Engineer at DocuSign[1], where his team rebuilds 50 million user product (DocuSign web app) using the tech stack of Node.js, Express.js, Backbone.js, CoffeeScript, Jade, Stylus and Redis.

Recently, he worked as an engineer at the curated social media news aggregator website, Storify.com[2] (acquired by LiveFyre[3]) which is used by BBC, NBC, CNN, The White House and others. Storify runs everything on Node.js unlike other companies. It's the maintainer of the open-source library jade-browser[4].

Before that, Azat worked as a CTO/co-founder at Gizmo[5] — an enterprise cloud platform for mobile marketing campaigns, and has undertaken the prestigious 500 Startups[6] business accelerator program.

Prior to this, Azat was developing he developed mission-critical applications for government agencies in Washington, DC, including the National Institutes of Health[7], the National Center for Biotechnology Information[8], and the Federal Deposit Insurance Corporation[9], as well as Lockheed Martin[10].

Azat is a frequent attendee at Bay Area tech meet-ups and hackathons (AngelHack[11] hackathon '12 finalist with team FashionMetric.com[12]).

[1]http://docusign.com
[2]http://storify.com
[3]http://livefyre.com
[4]http://npmjs.org/jade-browser
[5]http://www.crunchbase.com/company/gizmo
[6]http://500.co/
[7]http://nih.gov
[8]http://ncbi.nlm.nih.gov
[9]http://fdic.gov
[10]http://lockheedmartin.com
[11]http://angelhack.com
[12]http://fashionmetric.com

In addition, Azat teaches technical classes at General Assembly[13], Hack Reactor[14], pariSOMA[15] and Marakana[16] (acquired by Twitter) to much acclaim.

In his spare time, he writes about technology on his blog: webAppLog.com[17] which is number one[18] in "express.js tutorial" Google search results. Azat is also the author of Express.js Guide[19], Rapid Prototyping with JS[20] and Oh My JS[21]; and the creator of open-source Node.js projects, including ExpressWorks[22], mongoui[23] and HackHall[24].

Let's be Friends on the Internet

- Twitter: @RPJSbook[25] and @azat_co[26]
- Facebook: facebook.com/RapidPrototypingWithJS[27]
- Website: rapidprototypingwithjs.com[28]
- Blog: webapplog.com[29]
- GitHub: github.com/azat-co/rpjs[30]
- Storify: Rapid Prototyping with JS[31]

Other Ways to Reach Us

- Email: hi@rpjs.co[32]
- Google Group: rpjs@googlegroups.com[33] and https://groups.google.com/forum/#!forum/rpjs

Share on Twitter

> "I've finished reading the Rapid Prototyping with JS: Agile JavaScript Development book by @azat_co http://rpjs.co #nodejs #mongodb" — http://ctt.ec/4Vw73

This is a sample copy of Rapid Prototyping with JS: Agile JavaScript Development[34]. Get the full version on LeanPub in Kindle, ePub/iPad and PDF now, and **start shipping code that matters**!

[13]http://generalassemb.ly
[14]http://hackreactor.com
[15]http://parisoma.com
[16]http://marakana.com
[17]http://webapplog.com
[18]http://expressjsguide.com/assets/img/expressjs-tutorial.png
[19]http://expressjsguide.com
[20]http://rpjs.co
[21]http://leanpub.com/ohmyjs
[22]http://npmjs.org/expressworks
[23]http://npmjs.org/mongoui
[24]http://hackhall.com
[25]https://twitter.com/rpjsbook
[26]https://twitter.com/azat_co
[27]https://www.facebook.com/RapidPrototypingWithJS
[28]http://rapidprototypingwithjs.com/
[29]http://webapplog.com
[30]https://github.com/azat-co/rpjs
[31]https://storify.com/azat_co/rapid-prototyping-with-js
[32]mailto:hi@rpjs.co
[33]mailto:rpjs@googlegroups.com
[34]https://leanpub.com/rapid-prototyping-with-js