COMPLETED

JavaScript and Node FUNdamentals

A Collection of Essential Basics

//

AZAT MARDAN

JavaScript and Node FUNdamentals

A Collection of Essential Basics

Azat Mardan

This book is for sale at http://leanpub.com/jsfun

This version was published on 2014-05-29

Tweet This Book!

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

The suggested hashtag for this book is #JavaScriptFUNdamentals.

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=#JavaScriptFUNdamentals

Also By Azat Mardan

Contents

If it's not fun, it's not JavaScript.

1 ~~JavaScript FUNdamentals: The Powerful and Misunderstood Language of The Web~~

1.1 Expressiveness

Programming languages like BASIC, Python, C has boring machine-like nature which requires developers to write extra code that's not directly related to the solution itself. Think about line numbers in BASIC or interfaces, classes and patterns in Java.

On the other hand JavaScript inherits the best traits of pure mathematics, LISP, C# which lead to a great deal of expressiveness[1] (and fun!).

More about Expressive Power in this post: What does "expressive" mean when referring to programming languages?[2]

The quintessential Hello World example in Java (remember, Java is to JavaScript is what ham to a hamster):

```java
public class HelloWorld {
    public static void main(String[] args) {
        System.out.println("Hello World");
    }
}
```

The same example in JavaScript:

```javascript
console.log('Hello World')
```

or from within an HTML page:

[1] http://en.wikipedia.org/wiki/Expressive_power

[2] http://stackoverflow.com/questions/638881/what-does-expressive-mean-when-referring-to-programming-languages

```
<script>
document.write('Hello World')
</script>
```

JavaScript allows programmers to focus on the solution/problem rather that to jump through hoops and API docs.

1.2 Loose Typing

Automatic type casting works well most of the times. It a great feature that saves a lot of time and mental energy! There're only a few primitives types:

1. String
2. Number (both integer and real)
3. Boolean
4. Undefined
5. Null

Everything else is an object, i.e., mutable keyed collections. Read Stackoverflow on What does immutable mean?[3]

Also, in JavaScript there are String, Number and Boolean objects which contain helpers for the primitives:

```
'a' === new String('a') //false
```

but

```
'a' === new String('a').toString() //true
```

or

```
'a' == new String('a') //true
```

By the way, == performs automatic type casting while === not.

1.3 Object Literal Notation

Object notation is super readable and compact:

[3]http://stackoverflow.com/questions/3200211/what-does-immutable-mean

```
var obj = {
   color: "green",
   type: "suv",
   owner: {
      ...
   }
}
```

Remember that functions are objects?

```
var obj = function () {
   this.color: "green",
   this.type: "suv",
   this.owner: {
      ...
   }
}
```

1.4 Functions

Functions are **first-class citizens**, and we treat them as variables, because they are objects! Yes, functions can even have properties/attributes.

1.4.1 Create a Function

```
var f = function f () {
   console.log('Hi');
   return true;
}
```

or

```
function f () {
  console.log('Hi');
  return true;
}
```

Function with a property (remember functions are just object that can be invoked, i.e. initialized):

```
var f = function () {console.log('Boo');}
f.boo = 1;
f(); //outputs Boo
console.log(f.boo); //outputs 1
```

Note: the return keyword is optional. In case its omitted the function will return `undefined` upon invocation.

1.4.2 Pass Functions as Params

```
var convertNum = function (num) {
 return num + 10;
}

var processNum = function (num, fn) {
   return fn(num);
}

processNum(10, convertNum);
```

1.4.3 Invocation vs. Expression

Function definition:

```
function f () {};
```

Invocation:

```
f();
```

Expression (because it resolve to some value which could be a number, a string, an object or a boolean):

```
function f() {return false;}
f();
```

Statement:

```
function f(a) {console.log(a);}
```

1.5 Arrays

Arrays are also objects which have some special methods inherited from Array.prototype[4] global object. Nevertheless, JavaScript Arrays are **not** real arrays. Instead, they are objects with unique integer (usually 0-based) keys.

```
var arr = [];
var arr2 = [1, "Hi", {a:2}, function () {console.log('boo');}];
var arr3 = new Array();
var arr4 = new Array(1,"Hi", {a:2}, function () {console.log('boo');});
```

1.6 Prototypal Nature

There are **no classes** in JavaScript because objects inherit directly from other objects which is called prototypal inheritance: There are a few types of inheritance patterns in JS:

- Classical
- Pseudo-classical
- Functional

Example of the functional inheritance pattern:

```
var user = function (ops) {
  return { firstName: ops.name || 'John'
         , lastName: ops.name || 'Doe'
         , email: ops.email || 'test@test.com'
         , name: function() { return this.firstName + this.lastName}
         }
}

var agency = function(ops) {
  ops = ops || {}
  var agency = user(ops)
  agency.customers = ops.customers || 0
  agency.isAgency = true
  return agency
}
```

[4]https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/prototype#Properties

1.7 Conventions

Most of these conventions (with semi-colons being an exception) are stylistic, and highly preferential and don't impact the execution.

1.7.1 Semi-Colons

Optional semi-colons, except for two cases:

1. In for loop construction: `for (var i=0; i++; i<n)`
2. When a new line starts with parentheses, e.g., Immediately-Invoked Function Expression (IIFE): `;(function(){...}())`

1.7.2 camelCase

cameCase, except for class names which are CapitalCamelCase, e.g.,

```
var MainView = Backbone.View.extend({...})
var mainView = new MainView()
```

1.7.3 Naming

_,$ are perfectly legitimate characters for the literals (jQuery and Underscore libraries use them a lot).

Private methods and attributes start with _ (does nothing by itself!).

1.7.4 Commas

Comma-first approach

```
var obj = { firstName: "John"
          , lastName: "Smith"
          , email: "johnsmith@gmail.com"
          }
```

1.7.5 Indentation

Usually it's either tab, 4 or 2 space indentation with their supporters' camps being almost religiously split between the options.

1.7.6 White spaces

Usually, there is a space before and after `=`, `+`, `{` and `}` symbols. There is no space on invocation, e.g., `arr.push(1);`, but there's a space when we define an anonymous function: `function () {}`.

1.8 No Modules

At least until ES6[5], everything is in the global scope, a.k.a. `window` and included via `<script>` tags. However, there are external libraries that allow for workarounds:

- CommonJS[6]
- AMD and Require.js[7]

Node.js uses CommonJS-like syntax and **has** build-in support for modules.

To hide your code from global scope, make private attributes/methods use closures and immediately-invoked function expressions[8] (or IIFEs).

1.9 Immediately-Invoked Function Expressions (IIFEs)

```
(function () {
window.yourModule = {
...
};
}());
```

This snippet show an example of a object with private attribute and method:

[5]https://wiki.mozilla.org/ES6_plans
[6]http://www.commonjs.org/
[7]http://requirejs.org/
[8]http://en.wikipedia.org/wiki/Immediately-invoked_function_expression

```
(function () {
  window.boo = function() {
    var _a = 1;
    var inc = function () {
      _a++;
      console.log(_a);
      return _a;
    };
    return {
      increment: inc
    };
  }
}());
var b = window.boo();
b.increment();
```

Now try this:

```
b.increment();
b.increment();
b.increment();
```

1.10 Keyword "this"

Mutates/changes a lot (especially in jQuery)! Rule of thumb is to re-assign to a locally scoped variable before attempting to use `this` inside of a closure:

arrow function. perfect for the situation.

```
var app = this
$('a').click(function(e){
  console.log(this) //most likely the event or the      target anchor element
  console.log(app) //that's what we want!
  app.processData(e)
 })
```

When in doubt: console.log!

1.11 Pitfalls

JS is the only language that programmers think they shouldn't learn. Things like === vs. ==, global scope leakage, DOM, etc. might lead to problems down the road. This is why it's important to understand the language or use something like CoffeeScript, that take a way most of the issues.

1.12 Further Learning

If you liked this articled and would like to explore JavaScript more, take a look at this amazing **free** resource: Eloquent JavaScript: A Modern Introduction to Programming[9].

Of course for more advanced JavaScript enthusiasts and pros, there's my book Rapid Prototyping with JS[10] and intensive programming school HackReactor[11], where I teach part-time.

[9]http://eloquentjavascript.net/
[10]http://rpjs.co
[11]http://hackreactor.com

2 CoffeeScript FUNdamentals: The Better JavaScript

The CoffeeScript is a language that was built on top of JavaScript. CoffeeScript has some added benefits and its code is compiled into native JavaScript for execution. The CoffeeScript pros include: better syntax, function and class construction patterns, automatic `var` insertion, comprehensions and others.

Most of these perks will be obvious once we take a look at some examples. This quick language reference can get you started with CoffeeScript:

- Semicolons, Whitespace and Parentheses
- Vars
- Conditions
- Functions
- Classes
- Arrays
- Splats
- Comprehensions

2.1 Semicolons, Whitespace and Parentheses

While in JavaScript, semicolons are redundant and optional; in CoffeeScript they are banned.

The whitespace and indentation (typically two-space) are parts of the CoffeeScript language.

Parentheses for function invocation are optional (except when there are no arguments). The same goes for curly braces for object literals. We can even next objects without curly braces:

```
a =
  x: 1
  y: -20
  z: () ->
    console.log a.x+a.y

a.z()

b = [
  1,
  2,
    x: 10
    y: 20
]
```

Translates into this JavaScript:

```
var a, b;

a = {
  x: 1,
  y: -20,
  z: function() {
    return console.log(a.x + a.y);
  }
};

a.z();

b = [
  1, 2, {
    x: 10,
    y: 20
  }
];
```

As you might have noticed, the logical block's curly braces that we use to write code for functions (i.e., `{}`) are also replaced by indentation. Let's not forget that functions are just objects in JavaScript. :-)

2.2 Vars

CoffeeScript automatically inserts `var` keywords for us and prohibits manual usage of `var`. For example, `a`,`b`, and `c` variable declarations will have the `var` in the JavaScript code:

```
a = 10
b = 'x'
c = [1,2,3]
```

JavaScript code:

```
var a, b, c;

a = 10;

b = 'x';

c = [1, 2, 3];
```

CoffeeScript always puts `var`s at the top of the scope where this particular variable was encountered first. The scope is defined by the function or window. For example, the anonymous function `d` will have `e` scoped to it, because CoffeeScript first saw `e` inside of the function:

```
a = 10
b = 'x'
c = [1,2,3]

d = () ->
  e = a
  console.log e
d()
```

JavaScript output:

```
var a, b, c, d;

a = 10;

b = 'x';

c = [1, 2, 3];

d = function() {
  var e;
  e = a;
  return console.log(e);
};

d();
```

2.3 Conditions

Conditions are more readable by humans (English-like?) in CoffeeScript:

```
a = b = c = d = 1
if a is b or b isnt c and not c is d
  console.log 'true'
else
  console.log 'false'
```

```
var a, b, c, d;

a = b = c = d = 1;

if (a === b || b !== c && !c === d) {
  console.log('true');
} else {
  console.log('false');
}
```

So `is` is `===`, `isnt` is `!==`, `not` is `!`, `and` is `&&`, and `or` is `||`.

In CoffeeScript some fancy and arguably more readable constructions are possible:

```
console.log a if a is not null
if a isnt null then console.log a
if not a is null then console.log a
unless a is null then console.log a
```

Note: `unless` is just a shortcut for `if not`.

2.4 Functions

Functions in CoffeeScript are defined with arrows `()->` and and fat arrows `()=>` (more on this later):

```
a = (x,y) -> console.log x+y
a(10,-5)
```

JavaScript code:

```
var a;

a = function(x, y) {
  return console.log(x + y);
};

a(10, -5);
```

Longer expressions can be on multiple lines using indentation, while the default values can be assigned right in the function signature (i.e., `(name=value)`):

```
a = (x, y, z=15) ->
 sum = x + y + z
 console.log sum
a(10,-5)
```

```
var a;

a = function(x, y, z) {
  var sum;
  if (z == null) {
    z = 15;
  }
  sum = x + y + z;
  return console.log(sum);
};

a(10, -5);
```

So back to the far arrow, it does two things: 1. Defines a function 2. Binds the new function's scope to the current value of `this`

Remember that `this` is dynamically scoped, i.e., its meaning changes based on where it is situated in the code (what's the scope). For example, if we have a jQuery event handler `click`, we might want to use `this` as the object in which we defined the handler, not as the DOM element to which the handler is bound.

For example, this CoffeeScript code will return `window` object both times (that's what we want):

```
console.log @
$('div').click ()=>
  console.log @
```

The JavaScript code:

```
console.log(this);

$('div').click((function(_this) {
  return function() {
    return console.log(_this);
  };
})(this));
```

However, with single arrows it's back to the DOM scope for the event handler, (this might be bad if unexpected):

```
console.log @
$('div').click ()->
  console.log @
```

```
console.log(this);

$('div').click(function() {
  return console.log(this);
});
```

Traditionally for the snippet above, without the CoffeeScript's far arrows, you would see workarounds like these which use interim variables like `that`, `self`, or `_this`:

```
console.log(this);
var that = this;
$('div').click(function() {
  return console.log(that);
});
```

2.5 Classes

Classes are probably the most yummiest and the most complex and confusing feature in CoffeeScript. In JavaScript classes are absent at all! We use prototypes instead, so the objects inherit from other objects. We can also use factories, i.e., the functions that create objects.

However, if a developer wants to implement a class, it could be really tricky and often requires a good understanding of pseudo-classical instantiation patterns. This is not the case with CoffeeScript, which introduces `class` keyword. Inside of the `class` we can use `constructor` method and `super` call, for the initialization logic and the invocation of the parent's methods correspondingly.

For example, we have a parent class `Vehicle` from which we extend two classes `Compact` and `Suv`. In these classes, we write custom `move` methods with the `super` call, that allows us to re-use the logic from the parent class `Vehicle`.

```
class Vehicle
  constructor: (@name) ->

  move: (meters) ->
    console.log @name + " moved #{meters} miles."

class Compact extends Vehicle
  move: ->
    console.log "Cruising..."
    super 5

class Suv extends Vehicle
  move: ->
    console.log "Speeding..."
    super 45

camry = new Compact "Camry"
caddi = new Suv "Cadillac"

camry.move()
caddi.move()
```

The console outputs this:

```
Cruising...
Camry moved 5 miles.
Speeding...
Cadillac moved 45 miles.
```

The JavaScript output is quite lengthy, so no wonder developers often prefer functional or other patterns:

```
var Compact, Suv, Vehicle, caddi, camry,
  __hasProp = {}.hasOwnProperty,
  __extends = function(child, parent) { for (var key in parent) { if (__hasProp.c\
all(parent, key)) child[key] = parent[key]; } function ctor() { this.constructor \
= child; } ctor.prototype = parent.prototype; child.prototype = new ctor(); child\
.__super__ = parent.prototype; return child; };

Vehicle = (function() {
  function Vehicle(name) {
    this.name = name;
  }

  Vehicle.prototype.move = function(meters) {
    return console.log(this.name + (" moved " + meters + " miles."));
  };

  return Vehicle;

})();

Compact = (function(_super) {
  __extends(Compact, _super);

  function Compact() {
    return Compact.__super__.constructor.apply(this, arguments);
  }

  Compact.prototype.move = function() {
    console.log("Cruising...");
    return Compact.__super__.move.call(this, 5);
  };

  return Compact;

})(Vehicle);

Suv = (function(_super) {
  __extends(Suv, _super);

  function Suv() {
    return Suv.__super__.constructor.apply(this, arguments);
  }

  Suv.prototype.move = function() {
    console.log("Speeding...");
    return Suv.__super__.move.call(this, 45);
  };

  return Suv;

})(Vehicle);

```

```
camry = new Compact("Camry");

caddi = new Suv("Cadillac");

camry.move();

caddi.move();
```

2.6 Arrays and Slicing

Arrays in CoffeeScript can be defined just as they are in native JavaScript: `arr = [1, 2, 3]`. But we can do so much more with arrays in CoffeeScript! For example, we can use a range when defining an array (useful in iterators and comprehensions) and use slice:

```
arr = [1..10]
slicedArr = arr[2..4]
console.log arr, slicedArr
```

The console outputs:

```
[1, 2, 3, 4, 5, 6, 7, 8, 9, 10] [3, 4, 5]
```

```
var arr, slicedArr;

arr = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];

slicedArr = arr.slice(2, 5);

console.log(arr, slicedArr);
```

Trivia fact: for array declarations with 20+ items (e.g., range of `[0..20]` and larger), CoffeeScript compiler will switch to the `for` loop.

2.7 Splats

Splats is a better way of using a variable number of arguments and `arguments` object (from native JavaScript):

```
a = (x...) ->
 sum = 0
 x.forEach (item) -> sum += item
 console.log sum
a(10,-5, 15)
```

```
var a,
  __slice = [].slice;

a = function() {
  var sum, x;
  x = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
  sum = 0;
  x.forEach(function(item) {
    return sum += item;
  });
  return console.log(sum);
};

a(10, -5, 15);
```

Spats work with invocations too. For example, our `sum` function from the previous example needs to treat the array not as a first element, but as all `arguments`:

```
a = (x...) ->
 sum = 0
 x.forEach (item) -> sum += item
 console.log sum

a [-5..50]...
```

The output is `1260`. And the JavaScript:

```
var a, _i, _results,
  __slice = [].slice;

a = function() {
  var sum, x;
  x = 1 <= arguments.length ? __slice.call(arguments, 0) : [];
  sum = 0;
  x.forEach(function(item) {
    return sum += item;
  });
  return console.log(sum);
};

a.apply(null, (function() {
  _results = [];
  for (_i = -5; _i <= 50; _i++){ _results.push(_i); }
  return _results;
}).apply(this));
```

2.8 Comprehensions

The last but not least topic is comprehensions. They are probably the most used feature in CoffeeScript and replace (or at least try to replace) **all loops**.

For example, a simple iteration over an array:

```
arr = [
  'x',
  'y',
  'z'
]

for a in arr
  console.log a
```

The console output is:

```
x
y
z
```

The compiled code:

```
var a, arr, _i, _len;

arr = ['x', 'y', 'z'];

for (_i = 0, _len = arr.length; _i < _len; _i++) {
  a = arr[_i];
  console.log(a);
}
```

As is the case with conditions, comprehensions might be reversed in order, e.g., `console.log a for a in arr`. Then, we can get an index which will be the second parameter, e.g., `console.log a, i for a, i in arr` outputs:

```
x 0
y 1
z 2
```

The `when` clause acts like a `filter` method; in other words, we can apply a test to the iterator:

```
arr = ['x', 'y', 'z']
console.log a, i for a, i in arr when a isnt 'y'
```

The console outputs:

```
x 0
z 2
```

To step with an increment we can use `by`: `evens = (x for x in [0..10] by 2)`. In addition, for iterating over objects we can use `of`:

```
obj =
  'x': 10
  'y':-2
  'z': 50

coordinates = for key, value of obj
  "coordinate #{key} is #{value}pt"
console.log coordinates
```

The console output is:

```
["coordinate x is 10pt", "coordinate y is -2pt", "coordinate z is 50pt"]
```

The JavaScript code is:

```
var coordinates, key, obj, value;

obj = {
  'x': 10,
  'y': -2,
  'z': 50
};

coordinates = (function() {
  var _results;
  _results = [];
  for (key in obj) {
    value = obj[key];
    _results.push("coordinate " + key + " is " + value + "pt");
  }
  return _results;
})();
```

2.9 Conclusion

This CoffeeScript FUNdamentals is a concise overview that should highlight major pros of this language, which has many more useful features. We hope that classes, arrow function declaration, comprehensions, splats, and the clean syntax were enough to spark interest and lead to more exploration and experimentation with CoffeeScript.

Here's the list of further CoffeeScirpt reading:

- CoffeeScript Quirks[1]
- The Little Book on CoffeeScript[2]
- CoffeeScript Cookbook[3]
- Smooth CoffeeScript[4]
- CoffeeScript Ristretto[5]

[1]http://webapplog.com/coffeescript-quirks/
[2]http://arcturo.github.io/library/coffeescript/
[3]http://coffeescriptcookbook.com/
[4]http://autotelicum.github.io/Smooth-CoffeeScript/
[5]https://leanpub.com/coffeescript-ristretto/read

3 Backbone.js FUNdamentals: The Cornerstone of JavaScript MV* Frameworks

If you are reading this chapter, you're probably familiar with the benefits of using an MV* (asterisk means a controller, another view or something else) over plain jQuery that grows into unmanageable spaghetti code with time and added complexity.

Backbone.js is the cornerstone of all JavaScript frameworks because it's one of the most mature (i.e., dependable) and popular solutions in the ever-growing multitudes of browser JavaScript MV* (model-view-controllers and model-view-something) frameworks. If we draw a quick comparison between Backbone.js and other frameworks (not a fair one but still might be useful for some readers), it will be somewhere in the middle in complexity and features between Spine[1]&KnockoutJS[2] (the lighter side) and Ember.js[3]&Angular.js[4] (heavier side). Which one to choose depends on whether developers will have to customize a lot (go with lightweights) or use right-out-of-the-box (go with heavyweights).

A bit of trivia: Backbone.js was created by the same person (Jeremy Ashmenas[5]) that brought CoffeeScript and Underscore.js to the world!

Therefore, we'll demonstrate how to build a Backbone.js simple application from scratch. This is not your typical to-do list, examples of which you can find plenty at TodoMVC[6], but a simplified application that is easier to understand but that still uses all components: views, collections, subviews, models, and event binding. The example is the apple database application.

In this chapter we'll cover following:

- Typical Backbone.js App Structure
- Setting up Backbone.js App from Scratch
- Dependencies for the Backbone.js Project
- Working with Backbone.js Collections
- Event Binding with Backbone.js and jQuery
- Backbone.js View and Subviews with Backbone.js Underscore.js
- Super Simple Backbone Starter Kit
- Conclusion

[1]http://spinejs.com/

[2]http://knockoutjs.com/

[3]http://emberjs.com/

[4]https://angularjs.org/

[5]https://twitter.com/jashkenas

[6]http://todomvc.com/

3.1 Typical Backbone.js App Structure

There are just four classes in Backbone.js:

- Router: mapping of URL to methods
- View: methods related to browser&user events, HTML output (rendering)
- Collection: a list of models with extra helpers such as `fetch()`
- Model: an individual data item with extra helpers

A typical Backbone.js app will have the main Router class. From that class all the routes (URL paths) will be defined. The router can create and call methods of other classes such as Views, Collections, and Models. For example, for path `/books` we'll render books view.

The recommended usage is that Views usually have one Collection or one Model. A Collection is just a list of Models. However, this is not written in stone. Backbone.js is very flexible and should be used according to the specific needs. For example, an application can have a Model that has Views and each View that has collections.

In our concise example, we'll have one main Router that has Views. These Views will have models and collections.

3.2 Setting up Backbone.js App from Scratch

Firstly, we're going to build a typical starter "Hello World" application using Backbone.js and the model-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. The structure of the project is keep it simple stupid:

- `index.html`: the main file and all the application's JavaScript code
- `jquery.js`: jQuery library
- `underscore.js`: Underscore.js library
- `backbone.js`: Backbone.js library

A full source code for the "Hello World" app is available at GitHub under github.com/azat-co/rpjs/backbone/hello-world[7].

jQuery, Underscore are required dependencies for Backbone.js.

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

3.3 Dependencies for the Backbone.js Project

To build our apple database, we'll need to download the following libraries:

- jQuery 1.9 development source file[8]
- Underscore.js development source file[9]
- Backbone.js development source file[10]

Alternatively, you can hot-like these libraries from some CDNs (e.g., Google Hosted Libraries[11]), but then we'll need an internet connection every time you run your app.

Include these frameworks in the newly created `index.html` file like this:

```
<!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.

Some developers shy away from using Backbone Router (it's an optional class/component), but we always find that Router brings more benefits and clarity. In a way, it serves as a starting point of your application and helps to understand the foundation and how all the pieces fit together such as views, models, etc. So, let's define a simple Backbone.js Router inside of a `<script>` tag:

[8] http://code.jquery.com/jquery-1.9.0.js
[9] http://underscorejs.org/underscore.js
[10] http://backbonejs.org/backbone.js
[11] https://developers.google.com/speed/libraries/devguide

```
...
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 (the right part of the `route: action` key-value pair):

```
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({
});
```

Does it look familiar to you? Right, Backbone.js uses similar syntax for all of its components, such as `Backbone.View`, `Backbone.Router`, `Backbone.Model` and `Backbone.Collection`. The class is followed by the `extend` function and a JSON object as a parameter to it. This object often contains some initialization options or attributes of the class.

There are 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, because they allow us to do two things:

1. `el`: attach the Backbone View object to a Document Object Model (DOM) element
2. `template`: store the Underscore (in this case we use Underscore but it can be changed to another library) template

Here's the code for the home view:

```
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 home 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(){
      // some awesome code that will be executed during object's creation
    },
    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. The `app` variable is made global in this sense, this helps to access some Router properties (it's a good idea to use a prefix specific to your application):

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

Confused so far? Don't be because here is 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 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.

3.4 Working with Backbone.js Collections

Backbone Collections are useful classes that developers can use for storing any sets of data that belong to the same type. In other words, Collections are sets or lists of models, and they (collections) can have their own custom methods and logic as well as some built-in Backbone methods.

The full source code of Backbone Collections example is under the GitHub's rpjs/backbone/collections[12].

This example is about Backbone Collections, and it's built on top of the previous "Hello World" example from the **Setting up Backbone.js App from Scratch** which is available for download at rpjs/backbone/hello-world[13].

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"
  }
];
```

Grab the images (see picture below) from GitHub: https://github.com/azat-co/rpjs/tree/master/backbone/collections/img, or add your own.

Fuji apple illustration

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

Gala apple illustration

This is our apple *database*. :-) Or to be more accurate, our REST API endpoint-substitute, which provides us with names and image URLs of the apples (data models). If you want some real-world servers, you can use:

- Parse.com[14]: back-end-as-a-service provider
- Node.js and MongoDB REST API[15]: a free-JSON API built with Node.js, Express.js and MongoDB

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.

[14]http://parse.com

[15]http://webapplog.com/express-js-4-node-js-and-mongodb-rest-api-tutorial/

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:

```
  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.

So far, 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[16] folder, looks like this:

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

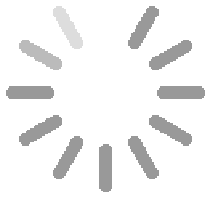
```
<!DOCTYPE>
<html>
<head>
  <script src="jquery.js"></script>
  <script src="underscore.js"></script>
  <script src="backbone.js"></script>

  <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.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!

3.5 Event Binding with Backbone.js and jQuery

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, see the picture below) to users to notify them that the information is being loaded.



Spinner GIF

It's a good thing that we have event binding in Backbone. The event binding is not an exclusive to Backbone feature, because it is part of jQuery. But without Backbone organization and code structure, things tend to end up messier (with plain jQuery). For example, 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 until 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[17].

[17]http://backbonejs.org/#View-constructor

```
  ...
  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[18]:

```
  ...
  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`:

[18] 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>
```

3.6 Backbone.js Views and Subviews with Underscore.js

This example is available at rpjs/backbone/subview[19].

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'
```

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

```
    },
    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.

```
  ...
  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));
  },
  ...
```

The `addToCart` method 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/>');
  },
...
```

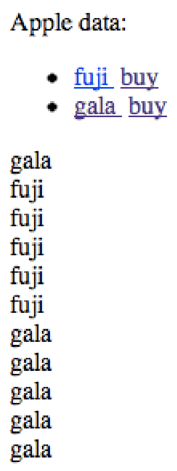
Finally, here is our long-awaited `render()` method, in which we iterate through each model in the collection (each apple), create an `appleItemView` for each apple, create an `<li>` element for each apple, and append that element to `view.listEl` — `<ul>` element with a class apples-list in the DOM:

```
...
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[20]:

```
<!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;
```

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

```
    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.

3.7 Super Simple Backbone Starter Kit

To jump-start your Backbone.js development, consider using Super Simple Backbone Starter Kit[21] by Azator[22], or similar projects like:

- Backbone Boilerplate[23]
- Sample App with Backbone.js and Twitter Bootstrap[24]
- More Backbone.js tutorials[25]

3.8 Conclusion

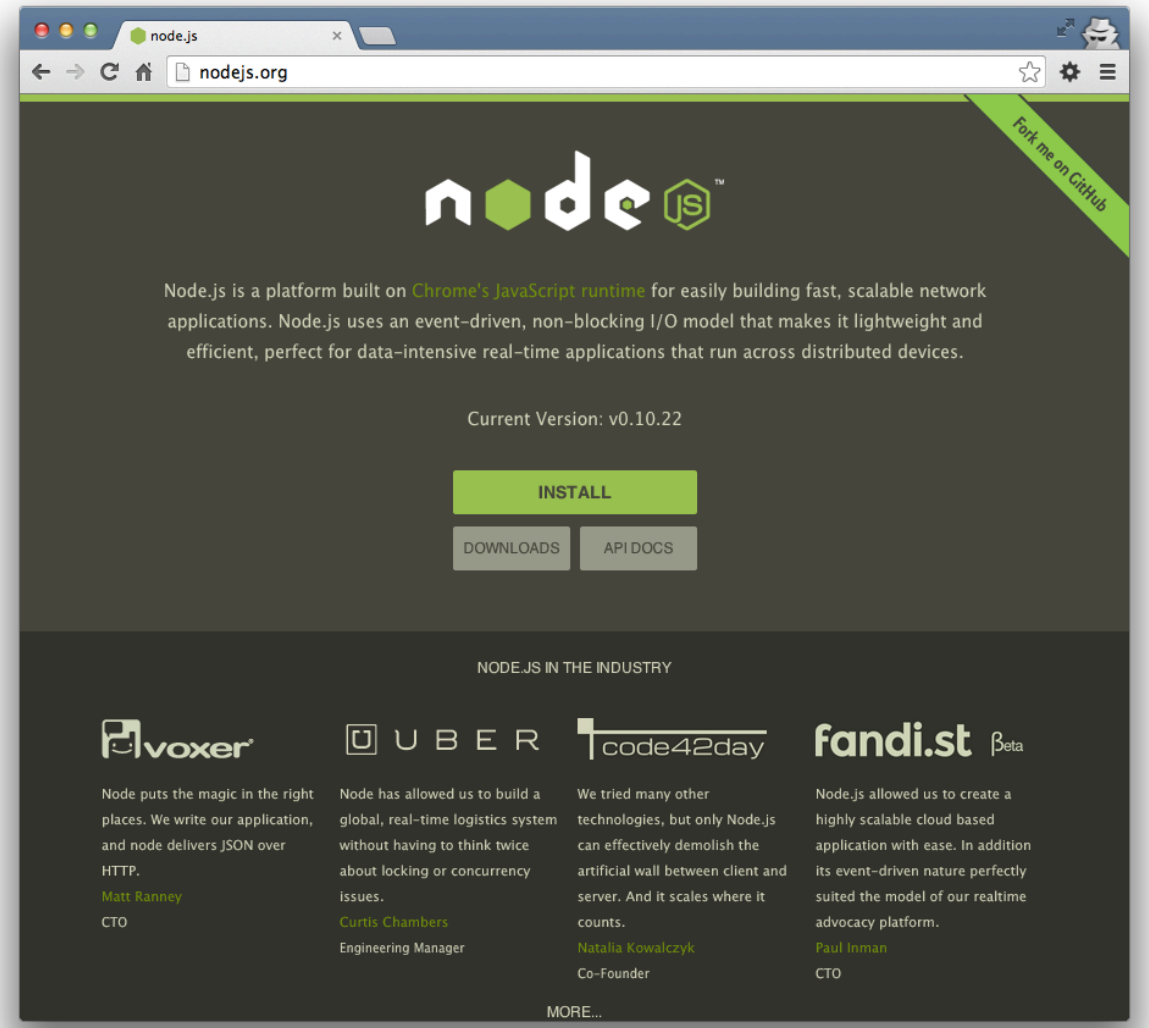
The already-mentioned in this chapter, and reputable Addy's[26] TodoMVC[27] resource contains **a lot** of browser JavaScript MV* frameworks with tutorials and examples. I often get asked "What do you think of X or Y?" The main point is not to get into analysis paralysis when choosing a front-end framework. Learn the foundation with Backbone.js, just because it's not so bloated and complicated, yet so widely-used and powerful in the right hands. Then pick a few newer and shinier libraries (e.g., React.js, Angular.js) and see how they fit the particular goals of project at hand.

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

4 ~~Node.js FUNdamentals: JavaScript on The Server~~

Node.js is a highly efficient and scalable non-blocking I/O platform that was build on top of Google Chrome V8 engine and its ECMAScript. This means that most front-end JavaScript (another implementation of ECMAScript) objects, functions and methods are available in Node.js. Please refer to JavaScript FUNdamentals[1] if you need a refresher on JS-specific basics.

[1]http://webapplog.com/js-fundamentals/



Node.js is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. Node.js uses an event-driven, non-blocking I/O model that makes it lightweight and efficient, perfect for data-intensive real-time applications that run across distributed devices.

Current Version: v0.10.22

NODE.JS IN THE INDUSTRY

Node puts the magic in the right places. We write our application, and node delivers JSON over HTTP.
Matt Ranney
CTO

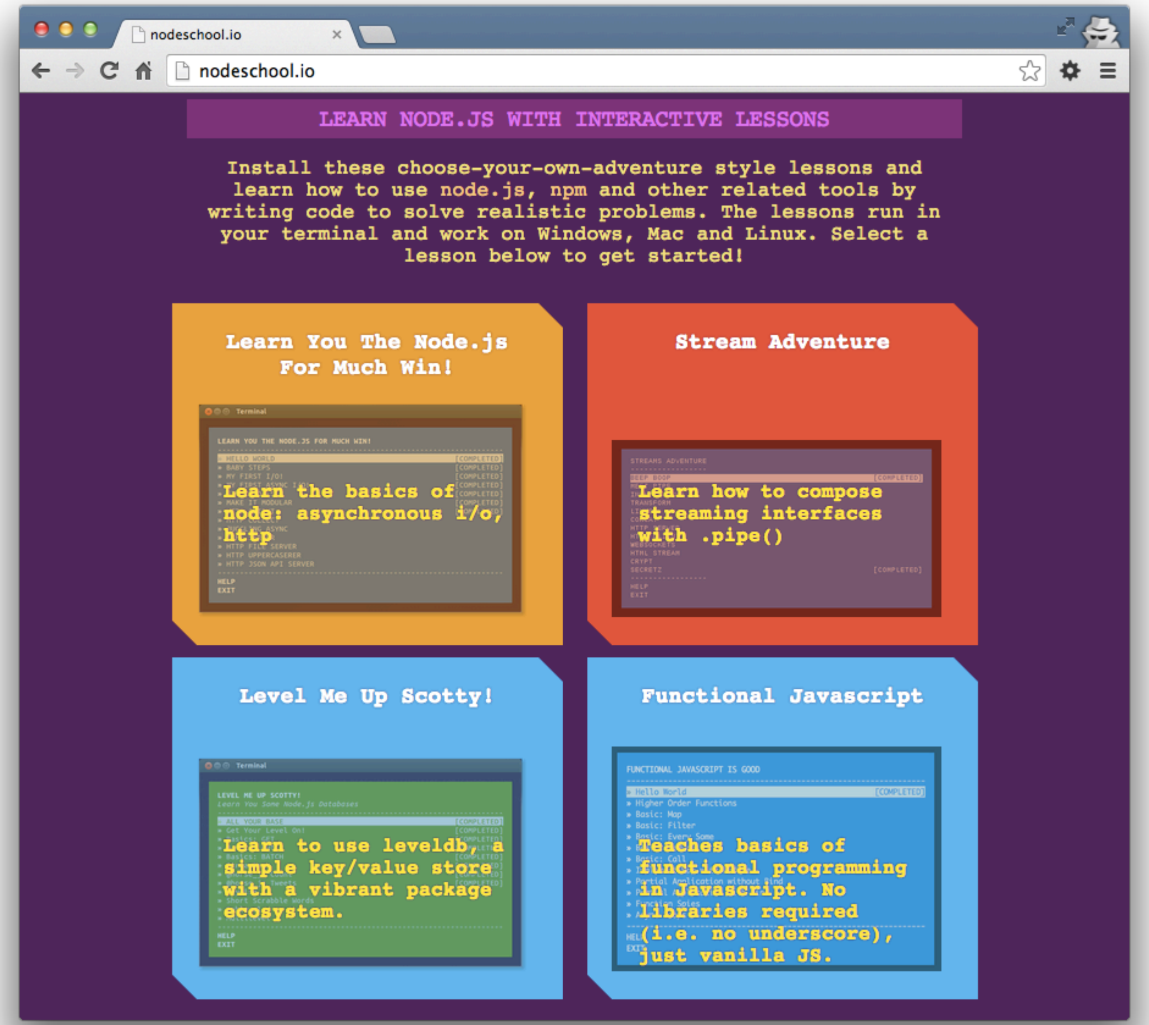
Node has allowed us to build a global, real-time logistics system without having to think twice about locking or concurrency issues.
Curtis Chambers
Engineering Manager

We tried many other technologies, but only Node.js can effectively demolish the artificial wall between client and server. And it scales where it counts.
Natalia Kowalczyk
Co-Founder

Node.js allowed us to create a highly scalable cloud based application with ease. In addition its event-driven nature perfectly suited the model of our realtime advocacy platform.
Paul Inman
CTO

Developers can install Node.js from its website[2] and follow this overview of main Node.js concepts. For more meticulous Node.js instructions, take a look at Rapid Prototyping with JS: Agile JavaScript Development[3] and Node School[4].

[2]http://nodejs.org
[3]http://rpjs.co
[4]http://nodeschool.io



LEARN NODE.JS WITH INTERACTIVE LESSONS

Install these choose-your-own-adventure style lessons and learn how to use node.js, npm and other related tools by writing code to solve realistic problems. The lessons run in your terminal and work on Windows, Mac and Linux. Select a lesson below to get started!

Learn You The Node.js For Much Win!

Learn the basics of node: asynchronous i/o, http

Stream Adventure

Learn how to compose streaming interfaces with .pipe()

Level Me Up Scotty!

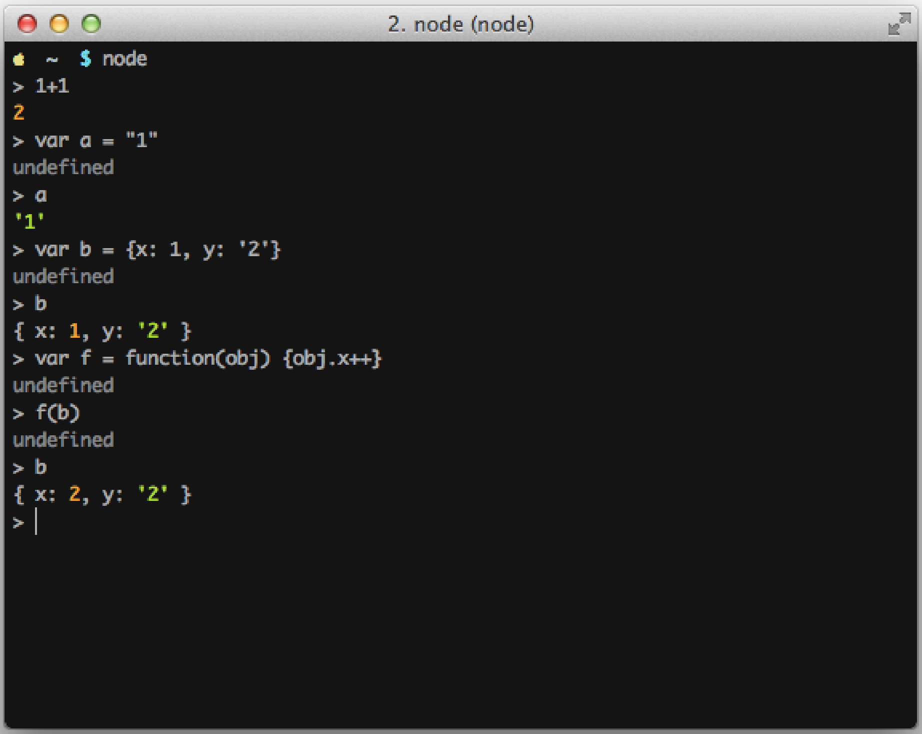
Learn to use leveldb, a simple key/value store with a vibrant package ecosystem.

Functional Javascript

Teaches basics of functional programming in Javascript. No libraries required (i.e. no underscore), just vanilla JS.

4.1 Read-Eval-Print Loop (a.k.a. Console) in Node.js

Like in many other programming language and platforms, Node.js has a read-eval-print loop tool which is opened by `$ node` command. The prompt changes to `>` and we can execute JavaScript akin to Chrome Developer Tools console. There are slight deviations in ECMAScript implementations in Node.js and browsers (e.g., `{}+{}`), but for the most part the results are similar.



```
~ $ node
> 1+1
2
> var a = "1"
undefined
> a
'1'
> var b = {x: 1, y: '2'}
undefined
> b
{ x: 1, y: '2' }
> var f = function(obj) {obj.x++}
undefined
> f(b)
undefined
> b
{ x: 2, y: '2' }
> |
```

So as you see, we can write JavaScript in the console all day long, but sometime we can to save script so we can run them later.

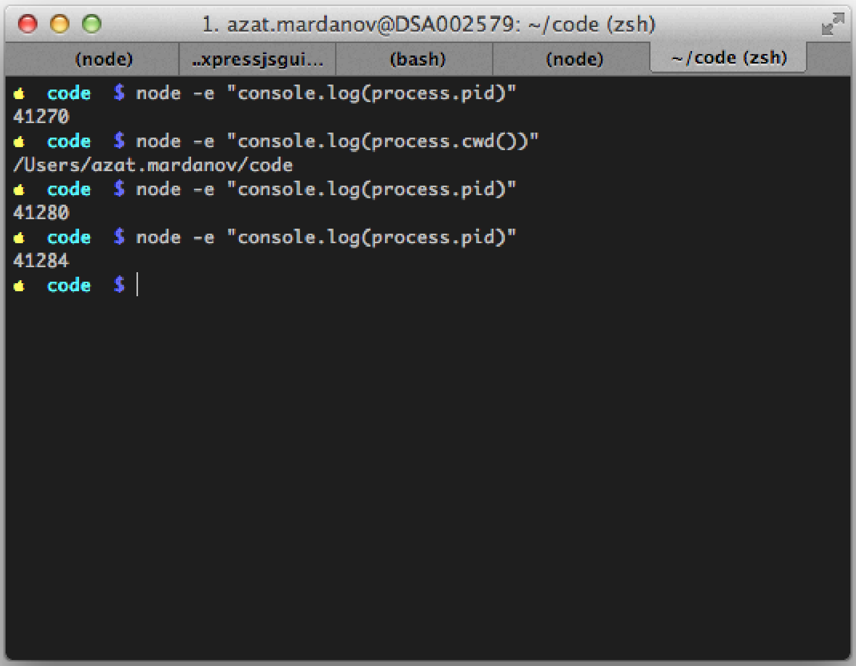
4.2 Launching Node.js Scripts

To start a Node.js script from a file, simply run `$ node filename`, e.g., `$ node program.js`. If all we need is a quick set of statements, there's a `-e` option that allow to run inline JavaScript/Node.js, e.g., `$ node -e "console.log(new Date());"`.

ps = process status

4.3 Node.js Process Information

Each Node.js script that is running is a process in its essence. For example, `ps aux | grep 'node'` will output all Node.js programs running on a machine. Conveniently, developers can access useful process information in code with `process` object, e.g., `node -e "console.log(process.pid)"`:



```
code  $ node -e "console.log(process.pid)"
41270
code  $ node -e "console.log(process.cwd())"
/Users/azat.mardanov/code
code  $ node -e "console.log(process.pid)"
41280
code  $ node -e "console.log(process.pid)"
41284
code  $ 
```

Node.js process examples using pid and cwd.

4.4 Accessing Global Scope in Node.js

As you know from JS FUNdamentals[5], browser JavaScript by default puts everything into its global scope. This was coined as one of the bad part of JavaScript in Douglas Crockford's famous [JavaScript: The Good Parts]. Node.js was designed to behave differently with everything being local by default. In case we need to access globals, there is a `global` object. Likewise, when we need to export something, we should do so explicitly.

In a sense, `window` object from front-end/browser JavaScript metamorphosed into a combination of `global` and `process` objects. Needless to say, the `document` object that represent DOM of the webpage is nonexistent in Node.js.

4.5 Exporting and Importing Modules

Another *bad part* in browser JavaScript is that there's no way to include modules. Scripts are supposed to be linked together using a different language (HTML) with a lacking dependency

[5]http://webapplog.com/

management. CommonJS[6] and RequireJS[7] solve this problem with AJAX-y approach. Node.js borrowed many things from the CommonJS concept.

To export an object in Node.js, use `exports.name = object;`, e.g.,

```
var messages = {
  find: function(req, res, next) {
  ...
  },
  add: function(req, res, next) {
  ...
  },
  format: 'title | date | author'
}
exports.messages = messages;
```

While in the file where we import aforementioned script (assuming the path and the file name is `route/messages.js`):

```
var messages = require('./routes/messages.js');
```

However, sometime it's more fitting to invoke a constructor, e.g., when we attach properties to Express.js app (more on Express.js in Express.js FUNdamentals: An Essential Overview of Express.js[8]). In this case `module.exports` is needed:

```
module.exports = function(app) {
  app.set('port', process.env.PORT || 3000);
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
  return app;
}
```

exports points to module.exports.

In the file that includes the example module above:

[6] http://www.commonjs.org/
[7] http://requirejs.org/
[8] http://webapplog.com/express-js-fundamentals/

```
...
var app = express();
var config = require('./config/index.js');
app = config(app);
...
```

The more succinct code: `var = express(); require('./config/index.js')(app);`.

The most common mistake when including modules is a wrong path to the file. For core Node.js modules, just use the name without any path, e.g., require('name'). Same goes for modules in `node_modules` folder. More on that later in the NPM section.

For all other files, use `.` with or without a file extension, e.g.,

```
var keys = require('./keys.js'),
  messages = require('./routes/messages.js');
```

In addition, for the latter category it's possible to use a longer looking statements with `__dirname` and `path.join()`, e.g., require(path.join(__dirname, ,'routes', 'messages'));'

If `require()` points to a folder, Node.js will attempt to read `index.js` file in that folder.

or main attribute in package.json

4.6 Buffer is a Node.js Super Data Type

Buffer is a Node.js addition to four primitives (boolean, string, number and RegExp) and all-encompassing objects (array and functions are also objects) in front-end JavaScript. We can think of buffers as extremely efficient data stores. In fact, Node.js will try to use buffers any time it can, e.g., reading from file system, receiving packets over the network.

4.7 __dirname vs. process.cwd

good to know.

`__dirname` is an absolute path to the file in which this global variable was called, while `process.cwd` is an absolute path to the process that runs this script. The latter might not be the same as the former if we started the program from a different folder, e.g., `$ node ./code/program.js`.

4.8 Handy Utilities in Node.js

Although, the core of the Node.js platform was intentionally kept small it has some essential utilities such as

- URL[9]
- Crypto[10]
- Path[11] querystring.
- String Decoder[12]

The method that we use in this tutorials is `path.join` and it concatenates path using an appropriate folder separator (/ or \\).

4.9 Reading and Writing from/to The File System in Node.js

Reading from files is done via the core `fs` module[13]. There are two sets of methods: async and sync. In most cases developers should use async methods, e.g., fs.readFile[14]:

```
var fs = require('fs');
var path = require('path');
fs.readFile(path.join(__dirname, '/data/customers.csv'), {encoding: 'utf-8'}, fun\
ction (err, data) {
  if (err) throw err;
  console.log(data);
});
```

And the writing to the file:

```
var fs = require('fs');
fs.writeFile('message.txt', 'Hello World!', function (err) {
  if (err) throw err;
  console.log('Writing is done.');
});
```

4.10 Streaming Data in Node.js

Streaming data is a term that mean an application processes the data while it's still receiving it. This is useful for extra large datasets, like video or database migrations.

Here's a basic example on using streams that output the binary file content back:

[9] http://nodejs.org/api/url.html
[10] http://nodejs.org/api/crypto.html
[11] http://nodejs.org/api/path.html
[12] http://nodejs.org/api/string_decoder.html
[13] http://nodejs.org/api/fs.html
[14] http://nodejs.org/api/fs.html#fs_fs_readfile_filename_options_callback

```
var fs = require('fs');
fs.createReadStream('./data/customers.csv').pipe(process.stdout);
```

By default, Node.js uses buffers for streams.

For a more immersive training, take a loot at stream-adventure[15] and Stream Handbook[16].

4.11 Installing Node.js Modules with NPM

NPM comes with the Node.js platform and allows for seamless Node.js package management. The way `npm install` work is similar to Git in a way how it traverses the working tree to find a current project[17]. For starter, keep in mind that we need either the `package.json` file or the `node_modules` folder, in order to install modules locally with `$ npm install name`, for example `$ npm install superagent`; in the program.js: `var suparagent = requier('superagent');`.

The best thing about NPM is that it keep all the dependencies local, so if module A uses module B v1.3 and module C uses module B v2.0 (with breaking changes comparing to v1.3), both A and C will have their own localized copies of different versions of B. This proves to be a more superior strategy unlike Ruby and other platforms that use global installations by default.

The best practice is **not to include** a `node_modules` folder into Git repository when the project is a module that supposed to be use in other application. However, it's recommended **to include** `node_-modules` for deployable applications. This prevents a breakage caused by unfortunate dependency update.

Note: The NPM creator like to call it `npm` (lowercase[18]).

4.12 Hello World Server with HTTP Node.js Module

Although, Node.js can be ~~use~~ used for a wide variety of tasks, it's mostly ~~knows~~ known for building web applications. Node.js is thrives in the network due to its asynchronous nature and build-in modules such as net and http.

Here's a quintessential Hello World examples where we create a server object, define request handler (function with req and res arguments), pass some data back to the recipient and start up the whole thing.

[15] http://npmjs.org/stream-adventure
[16] https://github.com/substack/stream-handbook
[17] https://npmjs.org/doc/files/npm-folders.html
[18] http://npmjs.org/doc/misc/npm-faq.html#Is-it-npm-or-NPM-or-Npm

```
var http = require('http');
http.createServer(function (req, res) {
  res.writeHead(200, {'Content-Type': 'text/plain'});
  res.end('Hello World\n');
}).listen(1337, '127.0.0.1');
console.log('Server running at http://127.0.0.1:1337/');]
```

[Function]require('events'). EventEmitter

The req and res parameters have all the information about a given HTTP request and response correspondingly. In addition, req and res can be used as streams (look in the previous section).

4.13 Debugging Node.js Programs

The best debugger is `console.log()`, but sometime we need to see the call stack and orient ourselves in async code a bit more. To do that, put `debugger` statements in your code and use `$ node debug program.js` to start the debugging process[19]. For more developer-friendly interface, download node inspector[20].

4.14 Taming Callbacks in Node.js

Callbacks[21] are able to Node.js code asynchronous, yet programmers unfamiliar with JavaScript, who come from Java or PHP, might be surprised when they see Node.js code described on Callback Hell[22]:

```
fs.readdir(source, function(err, files) {
  if (err) {
    console.log('Error finding files: ' + err)
  } else {
    files.forEach(function(filename, fileIndex) {
      console.log(filename)
      gm(source + filename).size(function(err, values) {
        if (err) {
          console.log('Error identifying file size: ' + err)
        } else {
          console.log(filename + ' : ' + values)
          aspect = (values.width / values.height)
          widths.forEach(function(width, widthIndex) {
```

[19]http://nodejs.org/api/debugger.html
[20]https://github.com/node-inspector/node-inspector
[21]https://github.com/maxogden/art-of-node#callbacks
[22]http://callbackhell.com/

```
            height = Math.round(width / aspect)
            console.log('resizing ' + filename + 'to ' + height + 'x' + height)
            this.resize(width, height).write(destination + 'w' + width + '_' + fi\
lename, function(err) {
              if (err) console.log('Error writing file: ' + err)
            })
          }.bind(this))
        }
      })
    })
  }
})
```

There's nothing to be afraid of here as long as two-space indentation is used. ;-) However, callback code can be re-write with the use of event emitters, promises or by utilizing the async library.

event driven

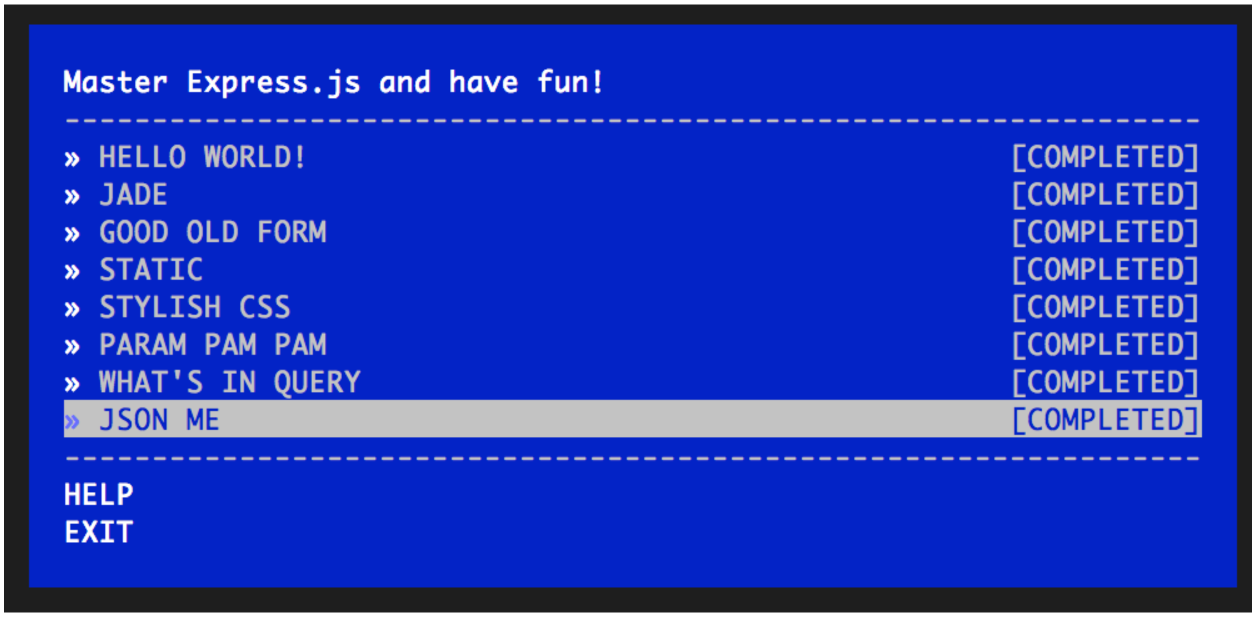
4.15 Introduction to Node.js with Ryan Dahl

Last, but not least, take a few minutes and watch this Introduction to Node.js with Ryan Dahl video[23].

4.16 Moving Forward with Express.js

After you've mastered Node.js basics in this article, you might want to read Express.js FUNdamentals: An Essential Overview of Express.js[24] and consider working on an interactive class[25] about the Express.js framework which is as of today is the most popular module on NPM.

[23]http://www.youtube.com/embed/jo_B4LTHi3I
[24]http://webapplog.com/express-js-fundamentals
[25]http://webapplog.com/expressworks



```
Master Express.js and have fun!
-----------------------------------------------------------------
» HELLO WORLD!                                       [COMPLETED]
» JADE                                               [COMPLETED]
» GOOD OLD FORM                                      [COMPLETED]
» STATIC                                             [COMPLETED]
» STYLISH CSS                                        [COMPLETED]
» PARAM PAM PAM                                      [COMPLETED]
» WHAT'S IN QUERY                                    [COMPLETED]
» JSON ME                                            [COMPLETED]
-----------------------------------------------------------------
HELP
EXIT
```

5 ~~Express.js FUNdamentals: The Most Popular Node.js Framework~~

Express.js is an amazing framework for Node.js projects and used in the majority of such web apps. Unfortunately, there's a lack of tutorials and examples on how to write good production-ready code. To mitigate this need, we released Express.js Guide: The Comprehesive Book on Express.js[1]. However, all things start from basics, and for that reason we'll give you a taste of the framework in this post, so you can decide if you want to continue the learning further.

5.1 Express.js Installation

Assuming you downloaded and installed Node.js (and NPM with it), run this command:

```
$ sudo npm install -g express@3.4.3
```

5.2 Express.js Command-Line Interface

Now we can use command-line interface (CLI) to spawn new Express.js apps:

```
$ express -c styl expressfun
$ cd expressfun && npm install
$ node app
```

Open browser at http://localhost:3000.

Here is the full code of `expressfun/app.js` if you don't have time to create an app right now:

[1]http://expressjsguide.com

```
var express = require('express');
var routes = require('./routes');
var user = require('./routes/user');
var http = require('http');
var path = require('path');

var app = express();

// all environments
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));

// development only
if ('development' == app.get('env')) {
  app.use(express.errorHandler());
}

app.get('/', routes.index);
app.get('/users', user.list);

http.createServer(app).listen(app.get('port'), function(){
  console.log('Express server listening on port ' + app.get('port'));
});
```

5.3 Routes in Express.js

If you open the `expressfun/app.js`, you'll see two routes in the middle:

```
...
app.get('/', routes.index);
app.get('/users', user.list);
...
```

The first one is basically takes care of all the requests to the home page, e.g., `http://localhost:3000/` and the latter of requests to `/users`, such as `http://localhost:3000/users`. Both of the routes process URLs case insensitively and in a same manner as with trailing slashes.

The request handler itself (`index.js` in this case) is straightforward: every thing from the HTTP request is in `req` and write results to the response in `res`:

```
exports.list = function(req, res){
  res.send("respond with a resource");
};
```

5.4 Middleware as The Backbone of Express.js

Each line above the routes is a middleware:

layered structure.

```
app.use(express.favicon());
app.use(express.logger('dev'));
app.use(express.bodyParser());
app.use(express.methodOverride());
app.use(app.router);
app.use(express.static(path.join(__dirname, 'public')));
```

request.on('data', cb); to get POST data.

The middleware is a pass thru functions that add something useful to the request as it travels along each of them, for example `req.body` or `req.cookie`. For more middleware writings check out Intro to Express.js: Parameters, Error Handling and Other Middleware[2].

5.5 Configuration of an Express.js App

Here is how we define configuration in a typical Express.js app:

```
app.set('port', process.env.PORT || 3000);
app.set('views', __dirname + '/views');
app.set('view engine', 'jade');
```

An ordinary settings involves a name (e.g., `views`) and a value (e.g., path to the folder where out templates/views live). There are more than one way to define a certain settings, e.g, `app.enable` for boolean flags.

5.6 Jade is Haml for Express.js/Node.js

Jade template engine is akin to Ruby on Rails' Haml in the way it uses whitespace and indentation, e.g., `layout.jade`:

[2]http://webapplog.com/intro-to-express-js-parameters-error-handling-and-other-middleware/

```
doctype 5
html
  head
    title= title
    link(rel='stylesheet', href='/stylesheets/style.css')
  body
    block content
```

Other than that, it's possible to utilize full-blown JavaScript code inside of Jade templates.

5.7 Conclusion About The Express.js Framework

As you've seen, it's effortless to create MVC web apps with Express.js. The framework is splendid for REST APIs as well. If you interested in them, visit the Tutorial: Node.js and MongoDB JSON REST API server with Mongoskin and Express.js[3] and Intro to Express.js: Simple REST API app with Monk and MongoDB[4].

If you want to know what are the other middlewares and configurations, check out Express.js API docs[5], Connect docs[6] and of course our book â€" Express.js Guide[7]. For those who already familiar with some basics of Express.js, I recommend going through ExpressWorks[8] â€" an automated Express.js workshop.

5.8 Update

For migrating from Express.js 3.x to 4.x take a look at this guide: Migrating Express.js 3.x to 4.x: Middleware, Route and Other Changes[9].

[3]http://webapplog.com/tutorial-node-js-and-mongodb-json-rest-api-server-with-mongoskin-and-express-js/
[4]http://webapplog.com/intro-to-express-js-simple-rest-api-app-with-monk-and-mongodb/
[5]http://expressjs.com/api.html
[6]http://www.senchalabs.org/connect/
[7]http://expressjsguide.com
[8]http://webapplog.com/expressworks
[9]http://webapplog.com/migrating-express-js-3-x-to-4-x-middleware-route-and-other-changes/

6 About the Author

Azat Mardan: A software engineer, author and 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]). Before that, Azat worked as a CTO/co-founder at Gizmo[4] — an enterprise cloud platform for mobile marketing campaigns, and he has undertaken the prestigious 500 Startups[5] business accelerator program. Previously, he was developing mission-critical applications for government agencies in Washington, DC: National Institutes of Health[6], National Center for Biotechnology Information[7], Federal Deposit Insurance Corporation[8], and Lockheed Martin[9]. Azat is a frequent attendee at Bay Area tech meet-ups and hackathons (AngelHack[10], and was a hackathon '12 finalist with team FashionMetric.com[11]).

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

In addition, Azat teaches technical classes at General Assembly[12] and Hack Reactor[13], pariSOMA[14] and Marakana[15] (acquired by Twitter) to much acclaim.

In his spare time, he writes about technology on his blog: webAppLog.com[16] which is number one[17] in "express.js tutorial" Google search results. Azat is also the author of Express.js Guide[18], Rapid Prototyping with JS[19] and Oh My JS[20].

6.1 Errata

Please help us make this book better by submitting issues via other means of communication listed below in the **Contact Us** section.

6.2 Contact Us

Let's be friends on the Internet!

- Tweet Node.js question on Twitter: @azat_co[21]
- Follow Azat on Facebook: facebook.com/profile.php?id=1640484994[22]
- GitHub: github.com/azat-co[23]

Other Ways to Reach Us

- Email Azat directly: hi@azat.co[24]
- Google Group: rpjs@googlegroups.com[25] and https://groups.google.com/forum/#!forum/rpjs
- Blog: webapplog.com[26]
- HackHall[27]: community for hackers, hipsters and pirates

[12]http://generalassemb.ly
[13]http://hackreactor.com
[14]http://parisoma.com
[15]http://marakana.com
[16]http://webapplog.com
[17]http://expressjsguide.com/assets/img/expressjs-tutorial.png
[18]http://expressjsguide.com
[19]http://rpjs.co
[20]http://leanpub.com/ohmyjs
[21]https://twitter.com/azat_co
[22]https://www.facebook.com/profile.php?id=1640484994
[23]https://github.com/azat-co
[24]mailto:hi@azat.co
[25]mailto:rpjs@googlegroups.com
[26]http://webapplog.com
[27]http://hackhall.com

Share on Twitter with ClickToTweet link: http://clicktotweet.com/HDUx0, or just click:

> "I've finished JavaScript and Node.js FUNdamentals: A Collection of Essential Basics by @azat_co #nodejs https://leanpub.com/jsfun"[28]

[28]http://ctt.ec/VQcEb