

Appendix C

Introduction to Single Page Applications

This Appendix provides an introduction to Single-Page Applications, commonly referred to as SPAs. The first part of this Appendix briefly discusses the rationale for creating a single-page application in JavaScript. The XHR2 section contains three Ajax-related examples, starting with a generic Ajax code sample, followed by a jQuery-based Ajax code sample. The second part of this Appendix provides an overview of BackboneJS and Twitter Bootstrap. The third part of this Appendix contains an abbreviated introduction to Jade, MongoDB, Mongoose, and NodeJS. The final portion of this Appendix provides code for a minimalistic SPA that you can enhance with your own custom code.

What is an SPA?

Let's start with a whirlwind "review" of Web application architecture. First there were servers that delivered static pages to Web sites, which evolved into servers that could deliver dynamically generated Web pages (using languages such as PHP) to websites. Next came Ajax, which provides partial-page refresh functionality that is more efficient and also creates a better user experience. Then a collection of Comet-based solutions arose in order to make the communication between browsers and servers more efficient. More recently, HTML5 WebSockets provide bidirectional full duplex asynchronous communication between browsers and servers via a persistent TCP connection using the WebSocket protocol, which is an improvement over the HTTP protocol that is the basis for all earlier Web applications.

During this time, various strategies were devised for the division of code between the browser and the server. One approach involves a "thick server" where most of the operations (such as generating dynamic Web pages and user authentication/validation) are performed, along with a "thin client" (or browser) that renders a Web page from a server but with limited additional functionality.

Modern Web Architecture

Recently there has been a proliferation of JavaScript toolkits that enable people to develop Web applications with a “thick client” and thin server. Although there is a diversity of architectures (even among recent Web sites), many modern Web applications have the following architecture:

- DOM (contains read-only data)
- Models (contain the state and data of the application)
- Modules (small independent subsystems)
- Views (they observe model changes via notifications)

SPAs are based on the preceding characteristics of a modern Web application: they have a model layer that handles data and view layers that read data from models, so they can redraw the UI without involving a server round-trip, which can give SPAs a more "native" feel to them. Fewer round-trips reduce the load on the server, which is obviously another benefit.

However, keep in mind the following points. First, there is no “specification” for SPA, and that SPA is evolving and improving over time as a result of the contributions from people in the community. Second, some toolkits that have an SPA also use Ajax, which arguably breaks the “ideal” design of SPA. Thus, different toolkits (and the applications that are based on those toolkits) have different implementations of SPA, which in turn increases the likelihood that you will find a toolkit that is well-suited to your specific needs.

MVC and MV* Patterns

The MVC (Model-View-Controller) pattern is well documented (plenty of online information is available) and has been popular for years, but the concept of controllers can be absent from SPAs and from some JavaScript toolkits. "Traditional" Web applications use a page refresh in order to navigate between independent views, whereas JavaScript Web applications based on SPA can retrieve (via Ajax) data from a server and dynamically rendered data in different views in the same Web page. Routers are used for navigation purposes because URLs are not updated while navigating to different views in a SPA Web application.

Although controllers usually update the view when the model changes (and vice versa), most JavaScript MVC frameworks tend to differ from the MVC pattern in terms of their use of controllers. As designs evolve in a community, variants emerge based on individual preferences and experiences. Since no individual or organization promotes the idea of MVC, these variants are free to co-exist.

Contrary to what you might read in online articles, `BackboneJS` does not have an MVC pattern: controllers are absent in current versions (and note that the `Controller` object in early versions was renamed to the `Router` object in later versions). Some controller-like logic is located in `BackboneJS` views and also located in `BackboneJS` routers (and the latter are employed to manage application state). `BackboneJS` views receive notifications when there are model changes, and the views are updated accordingly. `BackboneJS` can most likely be best classified as having an MV* architecture.

In addition, there are other models, such as the MVP (Model-View-Presenter) pattern and MVVM (which is used in .NET frameworks and in `KnockoutJS`). MVP appears to be better suited to Web applications in which there are numerous large views, and the role of the `Presenter` overlaps with that of a controller.

You can learn more about MVC and MVP in this article:

<http://martinfowler.com/eaDev/uiArchs.html>

Generating Web Pages in SPAs

As you already know, many Web pages are generated dynamically on a server that sends those Web pages to browsers. In modern Web applications, one solution is to generate Web pages using client-side templating, and another solution is to use code (the more traditional approach). The distinction between server-rendered and client-rendered Web pages helps to reinforce the fact that the server renders a given Web page, but the *client* renders subsequent updates.

Handling Model-Related Events in SPAs

The two major options for handling model data changes are observables and event emitters, and there is little difference between these two approaches. When a change occurs, the code that is “bound” to that change event is triggered. For example, events are registered on objects:

```
MyApp.on('change', function() { ... });
```

whereas observers are attached through global names:

```
Framework.registerObserver(window.MyApp, 'change', function() { ... });
```

Observables usually have some type of name resolution system, where you use strings in order to refer to objects. A global name resolution system (where names are strings rather than directly accessing objects) is often added for observables in order to facilitate the use of observers, which can only be registered when the objects they refer to have been instantiated.

Client-Side Technologies for SPAs

In brief, client-side technologies for an SPA include toolkits such as jQuery, BackboneJS, and Jade. However, you can certainly use other toolkits, such as variants of BackboneJS (Spine, Vertebrae, and so forth), or an entirely different toolkit (such as EmberJS). In addition to Jade, there are other templating engines available, including Mustache and Handlebars. Perform an Internet search for information and also a feature comparison of these toolkits.

BackboneJS

Backbone.js is an open source JavaScript toolkit for developing structured Web applications, and its home page is here:

<http://Backbone.js.org/>

Although Backbone.js is not based on an MVC (Model-View-Controller) pattern, there are some similarities. Backbone.js provides *models* with key-value binding and custom events, *collections* with an API of enumerable functions, and *views* with declarative event handling, all of which is connected to your existing API over a RESTful JSON interface. Backbone.js is an open-source component of DocumentCloud, and it's available on GitHub under the MIT software license, where you can find the source code, an online test suite, an example application, a list of tutorials, and real-world projects that use Backbone.js.

Download `Backbone.js` from its GitHub repository:

<https://github.com/documentcloud/backbone/>

A Brief Introduction to BackboneJS

The following subsections give you a simple overview of models, views, collections, and routers in BackboneJS. After you have read this section, you can search for online tutorials that show you examples of BackboneJS applications.

What is a Model?

According to the authors of backbone:

"Models are the heart of any JavaScript application, containing the interactive data as well as a large part of the logic surrounding it: conversions, validations, computed properties, and access control."

Whenever you create a new instance of a model, the `initialize()` method is invoked (which is also the case when you instantiate a new view in BackboneJS). A simple example is shown in the following code block:

```
Vehicle = Backbone.Model.extend({
  initialize: function(){
    alert("I am a vehicle");
  }
});
var vehicle = new Vehicle;
```

You can set parameters in a model in one of two ways. One way is to use a constructor, as shown here:

```
var vehicle = new Vehicle({model: "Ford", make: "Mustang"});
```

Another way is to use the `set` method, as shown here:

```
vehicle.set({make: "Ford", model: "Mustang"});
```

After setting property/value pairs, you can retrieve their values as follows:

```
var make = vehicle.get("make"); // "Mustang"
var model = vehicle.get("model"); // "Ford"
```

You can also set default values in the following manner:

```

Vehicle = Backbone.Model.extend({
  defaults: {
    make:    "Ford",
    model:   "Mustang"
  },
  initialize: function(){
    alert("I am a vehicle");
  }
});

```

Model Changes

You can listen for changes to the model and execute code whenever a change is detected.

For example, here the new contents of the initialize method:

```

initialize: function(){
  alert("I am a vehicle");

  this.bind("change:model", function(){
    var model = this.get("model"); // 'Stewie Griffin'
    alert("Changed my model to " + model);
  });
}

```

There are various other features available for manipulating models that you can read in the documentation.

What is a View?

```

SearchView = Backbone.View.extend({
  initialize: function(){
    alert("A view of a vehicle.");
  }
});

```

```

// The initialize function is like a constructor in that
// it is always called when you instantiate a Backbone View
var search_view = new SearchView;

```

What is a Collection?

A `Collection` in BackboneJS is an ordered set of models, and they are useful because you can include methods inside of collections to fetch data from a server, prepare that data before returning the collection, and set sample collections for debugging/testing.

Moreover, you can add event listeners and attach views to collections, which is not the case for simple JavaScript arrays.

As a simple example, we can use the `Model Vehicle` that we created in the previous section in order to create a `Collection` in Backbone as follows:

```
var Cars = Backbone.Collection.extend({
  model: Vehicle
});

var vehicle1 = new Vehicle({ make: "Ford",  model: "Mustang"  });
var vehicle2 = new Vehicle({ make: "GM",    model: "Camaro"   });
var vehicle3 = new Vehicle({ make: "GM",    model: "Corvette" });

var myCars = new Cars([ vehicle1, vehicle2, vehicle3]);
console.log( myCars.models ); // [vehicle1, vehicle2, vehicle3]
```

What is a Router?

BackboneJS routers are used for mapping URL fragments in your application (e.g.,

mapping `example.com/*/foo` to `shareFoo`). Routing the URLs in your

applications via URL fragments makes them useful for applications that need URL

routing and history capabilities. Defined routers should contain at least one route and a

function to map to that particular route. Keep in mind that routes interpret anything after

"#" tag in the URL, and all links in your application should target either `"/action"` or

`#action"`.

As a simple example, the following code block defines a Backbone router:

```
var MyAppRouter = Backbone.Router.extend({
  routes: {
    // matches http://example.com/#you-can-put-anything-here
    "**actions": "defaultRoute"
  },
  defaultRoute: function( actions ){
    // The variable passed in matches the variable
    // that is in the route definition "actions"
    alert( actions );
  }
});
```

```
// Initiate the router
```

```
var appRouter = new MyAppRouter;
```

```
// Start Backbone history (required for bookmarkable URLs)
```

```
Backbone.history.start();
```

This concludes a bare-bones introduction to BackboneJS, and the next section provides some useful links with more detailed information.

Useful Links

A collection of BackboneJS tutorials is here:

<http://backbonetutorials.com/>

An example of using BackboneJS with jQuery Mobile:

<http://coenraets.org/blog/2012/03/using-backbone-js-with-jquery-mobile/>

<http://weblog.bocoup.com/organizing-your-backbone-js-application-with-modules/>

Backbone Boilerplate

Backbone-boilerplate is an open source JavaScript toolkit that provides “boilerplate” functionality (analogous to toolkits such as HTML5 Boilerplate), and its home page is here:

<https://github.com/tbranyen/backbone-boilerplate>

The set-up instructions for Backbone-boilerplate are available in the `readme.md` file that is included in this distribution:

<https://github.com/backbone-boilerplate/grunt-bbb>

The set-up is somewhat lengthy (and beyond the constraints of this Appendix), but it's worth exploring this toolkit if you plan to use BackboneJS extensively in your code.

Variations of BackboneJS

There are many variations of BackboneJS (which are also extensions of BackboneJS), some of which are listed here:

```
pine.js
joint.js
ligament.js
vertebrae.js
bones.js (provides server-side functionality)
hambone.js
shinbone.js
```

Space constraints in this Appendix preclude a discussion of these toolkits, but you can perform an Internet search to find information about these toolkits.

One other variant of BackboneJS is SpineJS, which uses controllers (even though it is based on BackboneJS). Controllers in SpineJS provide the "glue" in Web applications, and they provide the functionality of traditional controllers.

The following SpineJS code block that defines a Spine controller with three functions that synchronizes changes in models to update views (and vice versa). Compare this code with BackboneJS code blocks that you saw earlier in this Appendix:

```
// Controllers inherit from Spine.Controller:
var RecipesController = Spine.Controller.sub({
  init: function() {
```

```

        this.item.bind('update', this.proxy(this.render));
        this.item.bind('destroy', this.proxy(this.remove));
    },

    render: function(){
        // Handle templating
        this.replace($('#recipeTemplate').tpl(this.item));
        return this;
    },

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

```

EmberJS

In addition to BackboneJS, there are various toolkits available, and one of the most popular is EmberJS, whose home page is here:

<http://emberjs.com/>

EmberJS applications download everything that is required to run during the initial page load, which is consistent with SPA. One point to keep in mind is that EmberJS adopts an MVC pattern as well as a heavy emphasis on routers, and also a templating engine.

EmberJS focuses on the “heavy lifting” to handle some of the more burdensome JavaScript tasks. On the other hand, BackboneJS has a much more minimalistic approach (but it can be enhanced with Backbone Boilerplate).

As a simple illustration, the following code block shows you how to define JavaScript objects in EmberJS.

```

Person = Ember.Object.extend({
    greeting: function(msg) {
        alert(msg);
    }
});

```

```

    }
  });

var person = Person.create();
// alert message: My Name is Dave
person.greeting("My name is Dave");

var person = Person.create({
  name: "Jeanine Smith",
  greeting: function() {
    this.greeting("My name is " + this.get('name'));
  }
});

// alert message: My Name is Jeanine
person.greeting();
You can extend objects in EmberJS, as shown here:

```

```

var HappyPerson = Person.extend({
  greeting: function(msg) {
    this._super(greeting.toUpperCase());
  }
});

```

You can also add event listeners in EmberJS, as shown here:

```

person.addObserver('name', function() {
  // do something after a name change
});
person.set('name', 'Oswald');

```

An extensive comparison between BackboneJS and EmberJS is here:

<http://www.i-programmer.info/programming/htmlcss/4966-creating-web-apps-the-camera-api.html>

If BackboneJS and EmberJS are not a good fit for your needs, you can perform an Internet search for other toolkits. You can also find various online articles that discuss the merits of various toolkits, and sometimes you can find articles that provide a direct comparison of various toolkits. One point to keep in mind is that there is an abundance of toolkits available, so if time is a factor, you might need to limit your analysis to a small number of toolkits so that you can properly assess them.

Twitter Bootstrap

Twitter Bootstrap is an open source project for creating Web sites and Web applications. This toolkit contains HTML and CSS-based design templates for UI Controls (such as forms, buttons, charts, navigation, and so forth), as well as user interactions.

Currently Twitter Bootstrap is the most watched and forked repository in GitHub, and you can download it here:

<http://twitter.github.com/Bootstrap/>

<http://blog.getBootstrap.com/>

Twitter Bootstrap was designed primarily as a style guide to document best practices, and also to be for people diverse skill levels. Bootstrap supports new HTML5 elements and syntax, and you can use Bootstrap as a complete kit or to start something more complex.

Twitter Bootstrap was initially created for modern browsers, but it has expanded its support to include all major browsers (including IE7). In addition, Twitter Bootstrap 2 supports tablets and smartphones, and its responsive design means that its components are scaled according to a range of resolutions and devices, thereby providing a consistent experience. Moreover, Twitter Bootstrap provides custom-built jQuery plugins, and it's built on top of the LESS toolkit. Some features of Twitter Bootstrap 2.0 are tooltips, styled stateful buttons, more table and form styling, and an improved structure for its CSS source code (multiple files instead of a single monolithic file). Bootstrap also supports styling for tabs, navigation bars, form controls, navigation lists, labels, and others. If you prefer, you can create your own custom download if you do not want to download the full toolkit.

Bootstrap requires HTML5 doctype, which is specified with this snippet:

```
<!DOCTYPE html>
```

Twitter Bootstrap sets basic global display, typography, and link styles (located in `scaffolding.less` of the distribution):

- remove margin on the body
- set background-color: white; on the body
- use the `@baseFontFamily`, `@baseFontSize`, and `@baseLineHeight` attributes as the typographic base
- set the global link color via `@linkColor` and apply link underlines only on `:hover`

Keep in mind that Twitter Bootstrap 2 uses much of `Normalize.css` (which also powers HTML5 Boilerplate) instead of the 'reset' block in version 1 of Bootstrap.s

Additional details are available here:

<http://twitter.github.com/bootstrap/scaffolding.html>

One interesting Bootstrap feature is its support for progress bars using the CSS classes `.bar` and `.progress` that are available in Bootstrap. As an illustration, the following code block shows how to render bars with different colors based on custom attributes that start with the string `progress-`, as shown here:

```
<div class="progress progress-info" style="margin-bottom: 9px;">
  <div class="bar" style="width: 20%"></div>
</div>
<div class="progress progress-success" style="margin-bottom: 9px;">
  <div class="bar" style="width: 40%"></div>
</div>
<div class="progress progress-warning" style="margin-bottom: 9px;">
  <div class="bar" style="width: 60%"></div>
</div>
<div class="progress progress-danger" style="margin-bottom: 9px;">
  <div class="bar" style="width: 80%"></div>
</div>
```

Useful Links

The following links provide a sample application using Twitter Bootstrap, as well as a set of videos and tutorials:

<https://dev.twitter.com/blog/say-hello-to-Bootstrap-2>

<http://webdesign.tutsplus.com/tutorials/workflow-tutorials/twitter-Bootstrap-101-tabs-and-pills/>

<http://twitter.github.com/Bootstrap/examples.html>

An example of BackboneJS with Twitter Bootstrap:

<http://coenraets.org/blog/2012/02/sample-app-with-backbone-js-and-twitter-Bootstrap/>

The following open source project uses Bootstrap with SaSS instead of LESS:

<https://github.com/jlong/sass-twitter-Bootstrap>

A Minimalistic SPA

The remainder of this Appendix shows you how to set up several server-side toolkits for creating a very simple SPA. You will see “bare bones” code that provides a starting point for an SPA that enables users to see the books that their friends have read, along with other information, such as reviews, interesting book quotes, and so forth. This application uses the following technologies:

Jade

MongoDB

Mongoose

NodeJS

The next section of this Appendix provides a high-level description of the technologies in the preceding list that have not been discussed already, and the final section of this Appendix contains code blocks that will help you understand the logic of the SPA application. As a suggestion, include additional toolkits (such as BackboneJS and jQuery) in this SPA so that you can expand your skillset.

Jade

Jade is a popular templating language for NodeJS, and its home page is here:

<http://jade-lang.com/>

Download Jade here:

<https://github.com/visionmedia/jade#readme>

After you uncompress the Jade distribution, you can create a single Jade JavaScript file with the following command:

```
make jade.js
```

Note that you can also install Jade via npm (the package manager for NodeJS) as follows:

```
npm install jade
```

Jade Code Samples

Jade uses HTML tags to generate HTML elements, the “#” syntax to generate id attributes, and the “.” syntax to generate class attributes. Indentation specifies nesting of elements. For example, the following Jade code block:

```
html
  head
  body
    div
      div#myid
      div.myclass
    p my paragraph
```

generates the following HTML:

```
<html>
  <head></head>
  <body>
    <div></div>
    <div id="myid"></div>
    <div class="myclass"></div>
    <p>my paragraph</p>
  </body>
</html>
```

Jade supports interpolation, so you can pass data to a Jade document. For example, suppose you have this data:

```
{fname: john, lname: smith}
```

and that you have this Jade element:

```
div #user #{fname} #{lname}
```

then the result is this HTML element:

```
<div id="user">john smith</div>
```

Jade also supports comments (single and multi-line), conditional logic, a "case" statement, filters, and iteration.

As a more concrete example, Listing C.1 displays the contents of `layout.jade`, which generates a very basic HTML5 Web page.

Listing C.1 `layout.jade`

```
doctype 5
html
  head
    title= title
    link(rel='stylesheet', href='/css/style.css')
    script(src='/js/jquery-1.8.2.min.js')
    script(src='/js/client.js')
    script(src='/js/templates.js')
  body
    block content
```

As you can see in Listing C.1, Jade uses an easy syntax for defining links to CSS stylesheets and for defining HTML `<script>` elements.

In addition, Jade allows you to extend the contents of a template and also use conditional logic. Listing C.2 extends the contents with `layout.jade` based on whether or not `user.id` (which is defined elsewhere) is defined.

Listing C.2 `index.jade`

```
extends layout
```



```

block content
  h1= title
  - if (user.id)
    p Hello #{user.name}, and welcome to #{title}
    p
      a(href='/managebooks') My Book List
    p
      a(href='/logout') Logout
    #bookhistory
  - else
    div
      a(href='/login') Login with password
    div
      a(href='/register') Register

```

As you can see, the first portion of Listing C.2 (starting with “if”) displays a welcome message and a link where the currently logged in user can see a list of books. The second portion of Listing E.2 (starting with “else”) displays a `<div>` element with a link for the login page and a second `<div>` element for a registration page.

A Minimal NodeJS Code Sample with Jade

This section shows you how to create a tiny NodeJS application that uses Jade with the following three Jade files (which are placed in a `views` subdirectory) and one JavaScript file:

```

views/index.jade
views/layout.jade
views/navigation.jade
server.js

```

Here are the contents of `index.jade`:

```

h1
  a(href='http://www.google.com/') Google Home Page

```

Here are the contents of `layout.jade`:

```

!!! 5
html(lang='en')

```

```
    head
    title= title
    body!= body
    div#navigation!= partial('navigation.jade')
```

Here are the contents of navigation.jade:

```
div#navigation
  a(href='/') home
```

Listing C.3 displays the contents of `server.js` that references the three Jade files in this section.

Listing C.3 server.js

```
var express = require('express');
var app = express.createServer();

app.configure(function () {
  app.set('views', __dirname + '/views');
  app.set('view engine', 'jade');
});

app.get('/', function(req, res) {
  res.render('index.jade',
    { pageTitle: 'Jade Example', layout: false });
});

app.listen(9000);
```

Although we have not covered Node-related code, you can probably figure out that the code creates an application, sets some configuration values, and then renders the HTML Web page that is defined by the Jade-based templates in the view subdirectory.

Other Templating Solutions

There are many other templating solutions available, and some of the more popular ones are listed here:

```
Mustache.js
Handlebars.js
Dust.js
jQuery.tmpl plugin
Underscore Micro-templating
```

PURE

You can perform an Internet search to find links and tutorials with examples that illustrate how to use the preceding templating engines.

MongoDB

MongoDB is a popular NoSQL database, and its home page is here:

<http://www.mongodb.org/>

A good place to start working with MongoDB is here:

<https://mongolab.com/home>

Download and install MongoDB from this Web site:

<http://www.mongodb.org/downloads>

After you install MongoDB, you can start MongoDB with the following command:

```
mongodb
```

You can also launch an interactive session by typing `mongo` in another command shell, which enables you to manage schemas and perform CRUD-like operations from the command line.

For example, the following sequence of commands shows you how to insert a new user in the `users` schema (which is defined in the `Mongoose` section later in this Appendix).

```
mongo
>
> use db
switched to db db
> users = db.users
db.users
> users.find();
>
> users.insert({firstName:'a', lastName:'b'});
> users.find()
{
```

```
    "_id" : ObjectId("509425f82685d84f9c41c858"),
    "firstName" : "a",
    "lastName" : "b"
}
```

Consult the documentation for additional details about commands that you can execute in the MongoDB command shell.

NodeJS

NodeJS is a popular server technology that executes JavaScript. You can download and install NodeJS from its home page:

<http://nodejs.org/>

This section will only describe the basic information and the modules that you need in order to work with a NodeJS server that is used in the sample application on the CD-ROM

.

After you have installed NodeJS on your machine, install `express` and `mongoose` with these commands:

```
npm install express
npm install mongoose
```

Now you are ready to launch a NodeJS-based application. Listing C.4 displays the contents of `HelloWorld.js` that displays the text string “Hello World”.

Listing C.4 HelloWorld.js

```
//create an app server
var express = require("express");
var app = express();
```

Launch the application from the command line as follows:

```
node HelloWorld.js
```

Launch a browser and navigate to the URL <http://localhost:5000> and you will see the text string “Hello World”.

Now that you know how to launch an application with NodeJS, let's look at an example that uses Jade templates. Listing C.5 displays the contents of `server.js` that references the three Jade files that you saw in a previous section ("A Minimal SPA").

Listing C.5 `server.js`

```
//create an app server
var express = require("express");
var app = express();

//set path to the views (template) directory
app.set('views', __dirname + '/views');

//set path to static files
app.use(express.static(__dirname + '/../public'));

//handle GET requests on /
app.get('/', function(req, res){
    res.render('index.jade', {title: 'Google Home Page'});
});

//listen on localhost:5000
app.listen(5000);
```

The code in Listing C.5 is straightforward: define an `express` variable and an `app` variable, set the path to the `views` directory (which contains the Jade files that you saw in an earlier section), and then define an anonymous JavaScript function that references the Jade file `index.jade` when users launch a browser here:

```
http://localhost:5000
```

The final line of code in Listing C.4 specifies port 5000 on your local machine as the location where users can access this NodeJS-based application.

Finally, launch this Node-based application as follows:

```
node server.js
```

Although this example uses a file called `server.js`, you can choose any other valid filename.

Mongoose

Mongoose is an ORM (written in NodeJS) for MongoDB that enables you to define schemas in a MongoDB database, and its home page is here:

<http://mongoosejs.com/index.html>

Install Mongoose with the npm package manager as shown here:

```
npm install mongoose
```

The following sections show you how to connect to a MongoDB instance via Mongoose, followed by an example of creating schemas, creating instances of the schemas, and then modifying and saving those instances to a MongoDB database.

Connecting to MongoDB via Mongoose

You can connect to MongoDB with the `mongoose.connect()` method. The following command is the minimum needed to connect the myapp database running locally on the default port (27017):

```
mongoose.connect('mongodb://localhost/myapp');
```

You can specify additional parameters in the `uri` depending on your environment:

```
mongoose.connect('mongodb://username:password@host:port/database');
```

Consult the documentation for additional options for connecting to a MongoDB database.

Creating Schemas in Mongoose

The permitted SchemaTypes in Mongoose are: String, Number, Date, Buffer, Boolean, Mixed, ObjectId, and Array.

First we need to define two variables that reference Mongoose and a MongoDB database as follows:

```
var mongoose = require('mongoose'),
    db = mongoose.createConnection('localhost', 'test');
```

Now let's create a simple Person schema in Mongoose:

```
var personSchema = new mongoose.Schema({
  id:          personid,
  firstName: String,
  lastName:   String
})
```

We can add methods to our Person schema, as shown here:

```
personSchema.methods.pname = function () {
  var fullname = this.firstname+" "+this.lastname;
  console.log(fullname);
}
```

Next we compile our Person schema into a Model as follows:

```
var Person = db.model('Person', personSchema);
```

Note that a model is just a class that enables us to construct documents, and in this

example, a document is a Person object.

Let's create two Person objects as follows:

```
var p1 = new Person({ firstName: 'John', lastname: 'Smith' })
var p2 = new Person({ firstName: 'Jane', lastname: 'Jones' })
```

Finally, we can save our objects in MongoDB as follows:

```
p1.save(function (error) {
  if (error) {
    console.log("error saving p1");
  }
});

p2.save(function (error) {
  if (error) {
    console.log("error saving p2");
  }
});
```

You can specify other properties in a Mongoose schema, such as indexes, setters, and getters. For example, the following code block shows you how to expand the definition of the `User` schema with several additional properties:

```
mongoose.model('User', {
  properties: ['first', 'last'],
  cast:       {age: Number},
  indexes:    ['first', 'last'],
  setters:    {first: function(){} },
  getters:    {full_name: function(){} },
  methods:    {save: function() }
});
```

An SPA Code Sample

The SPA code sample creates three Mongoose schemas to represent users, books, and user comments about books.

The `userSchema` contains information about users and the list of books that each user had read, which has a one-to-many relationship (each user can read multiple books), as shown here:

```
var userSchema = new mongoose.Schema({
  userid:      Schema.Types.ObjectId,
  firstName:   String,
  lastName:    String,
  date:        {type: Date, default: Date.now},
  books:       [{type: Schema.Types.ObjectId, ref: 'Book'}]
});
```

The `bookSchema` contains information about books and the list of users that have read each book, which has a one-to-many relationship (each book can be read by multiple users), as shown here:

```
var bookSchema = new mongoose.Schema({
  bookid:      Schema.Types.ObjectId,
  bookAuthor:  String,
  bookTitle:   String,
  bookPubDate: {type: Date},
  bookEdition: Number,
  date:        {type: Date, default: Date.now},
  users:       [{type: Schema.Types.ObjectId, ref: 'User'}]
```



```
}}
```

The `commentSchema` contains information about the comments that users make about books that they have read, which references a user, a book, and a comment about a book, as shown here:

```
var commentSchema = new mongoose.Schema({
  commentid: Schema.Types.ObjectId,
  userid:    {type: Schema.Types.ObjectId, ref: 'User'},
  bookid:    {type: Schema.Types.ObjectId, ref: 'Book'},
  comment:   String,
  date:      {type: Date, default: Date.now}
})
```

Now we can compile our three schemas into Models as follows:

```
var Book      = db.model('Book', bookSchema);
var User      = db.model('User', userSchema);
var Comment   = db.model('Comment', commentSchema);
```

Finally, we can create objects for the three schemas. The following code block creates a new book, a new user, and a comment about the new book by the new user:

```
var b1 = new Book({bookAuthor: 'Oswald Campesato',
                  bookTitle:   'HTML5 Canvas and CSS3 Graphics',
                  bookPubDate: '07/30/2012'
                  });

var u1 = new User({firstName: 'John', lastname: 'Smith'});
var c1 = new Comment({userid: u1, bookid: b1, comment: "Great book!"});
```

Now we need to update `b1` to add user `u1` to the list of users who have read the book:

```
b1.users.push(u1);
```

Next, update `u1` to add the book `b1` to the list of books that user `u1` has read:

```
u1.books.push(b1);
```

Save the modified `b1`, `u1`, and `c1` with the following code snippet:

```
b1.save();
u1.save();
c1.save();
```

You also need code that enables users to find and update existing books and users. The following method enables us to find and update a book by its id value:

```
Book.findByIdAndUpdate(b1.bookid, { $set: { books: appendB1}},  
  function (err, book) {  
    if (err) return handleError(err);  
    res.send(book);  
  });
```

The following method enables us to find and update a user:

```
Book.findByIdAndUpdate(b1.userid, { $set: { books: appendU1}},  
  function (err, book) {  
    if (err) return handleError(err);  
    res.send(book);  
  });
```

Now you know how to do the following:

- connect to a MongoDB database in Mongoose
- create MongoDB schemas using Mongoose
- create and insert documents for a MongoDB database

The code in Listing C.6 contains the code (the schema definitions are shown above and are not displayed in the code sample) for connecting to a MongoDB instance, creating schemas, populating some instances of the schemas, and then saving the instances to the database.

Listing C.6 app.js

```
/* server */  
var express = require('express')  
  , app = express.createServer()  
  , mongoose = require('mongoose')  
  , db = mongoose.createConnection('localhost', 'test');  
  
/* models */  
mongoose.connect('mongodb://127.0.0.1/sampled');  
  
var Schema = mongoose.Schema  
  , ObjectId = Schema.ObjectID;  
  
// schemas declared earlier in Mongoose section  
var userSchema = ...  
var bookSchema = ...  
var commentSchema = ...  
  
var Book      = db.model('Book', bookSchema);
```

```

var User    = db.model('User', userSchema);
var Comment = db.model('Comment', commentSchema);

//A new book, a new user, and a comment
//about the new book by the new user:

var b1 = new Book({bookAuthor:  'Oswald Campesato',
                    bookTitle:    'HTML5 Canvas and CSS3 Graphics',
                    bookPubDate:  '07/30/2012'
                    });

var u1 = new User({firstName:  'John', lastname: 'Smith'});
var c1 = new Comment(
    {userid: u1, bookid: b1, comment: "Great book!"});

// Add user u1 to the list of users (in b1) who have read the book:
b1.users.push(u1);

// Add book b1 to the list of books (in u1) that user u1 has read:
u1.books.push(b1);

//show books
app.get('/showbooks', function(req,res){
console.log("finding books");
    Book.find({}, function(error, data){
        res.json(data);
    });
});

//show users
app.get('/showusers', function(req,res){
console.log("finding users");
    User.find({}, function(error, data){
        res.json(data);
    });
});

// sample URL for adding a user:
//http://localhost:3003/adduser/tom/smith/pasta

app.get('/adduser/:first/:last/:username', function(req, res){
console.log("adding user");
    var user_data = {
        first_name: req.params.first
        , last_name: req.params.last
        , username: req.params.username
    };

    var user = new User(user_data);

    user.save( function(error, data){
        if(error){
            res.json(error);
        }
    })

```

```

        else{
            res.json(data);
        }
    });
});

```

```

app.listen(3003);
console.log("listening on port %d", app.address().port);

```

Listing C.6 consolidates many of the code fragments that you saw earlier in this Appendix. There is also a code block that enables you to add new users by extracting the relevant pieces of information from a URL and then constructing the following structure that is saved to the database:

```

//sample URL for adding a user:
//http://localhost:3003/adduser/tom/smith/pasta

app.get('/adduser/:first/:last/:username', function(req, res){
    var user_data = {
        first_name: req.params.first
        , last_name: req.params.last
        , username: req.params.username
    };

    var user = new User(user_data);

    user.save( function(error, data){
        if(error){
            res.json(error);
        }
        else{
            res.json(data);
        }
    });
});

```

Now open a command shell and launch MongoDB with the following command:

```

mongod

```

Open a second shell and launch the code in Listing C.6 as follows:

```

node app.js

```

Summary

In this Appendix, you learned about the rationale for creating a single-page application (SPA). You also learned about technologies such as Jade, MongoDB, Mongoose, and NodeJS that you can use for supporting the different parts of an SPA.