JAVASCRIPT FRAMEWORK GUIDE

ANGULARJS, BACKBONE, EMBER

CONFIDENTLY CHOOSING AND QUICKLY LEARNING

BACKBONE.JS

CRAIG MCKEACHIE

JavaScript Framework Guide

AngularJS, Backbone, Ember

Confidently Choosing and Quickly Learning

Written by Craig McKeachie

First Edition

For my wife who not only supports me but lifts me higher.
And for my kids so you might take a divergent road and it make all the difference.

Table of Contents

Preface

Who is this book for?

This book is for the developer who loves to learn but is starting to have trouble keeping up. It's for the developer who wants to build ambitious and awesome web applications. This book helps you understand the big picture of what is happening with JavaScript frameworks. It's for the developer who values his time and understands that a book that costs less than what he or she makes in an hour and saves them hundreds of hours is a good value.

What does this book cover?

The focus of this book is helping you:

- Decide whether to use a JavaScript framework on your project
- Confidently choose a framework
- Learn the major frameworks

If you are reading this book, I assume you've built web applications with popular server-side technologies such as PHP, Rails, ASP.NET, Python, etc. And are now looking for information on how to build richer client applications in JavaScript.

To ask a question, make a comment, or purchase another copy of this book, visit funnyant.com.

CHAPTER 1

SINGLE-PAGE APPLICATIONS EXPLAINED

What is a Single-Page Application?

What exactly are we talking about when we say "single-page application" (SPA)? Let's begin with the end in mind. We want to create web applications that provides a more responsive and fluid user experience akin to a desktop application. Single-page applications (SPAs) are characterized by "the view the users sees" and the URL changing but the web browser never reloading the page.

In traditional web applications, where the entire page reloads frequently, it is common for the user to get lost and not know where he is in the application. This feeling is almost entirely eliminated in a single-page application. A more fluid experience is achieved by only loading small sections of a page into an existing page instead of replacing the entire page.

Just AJAX?

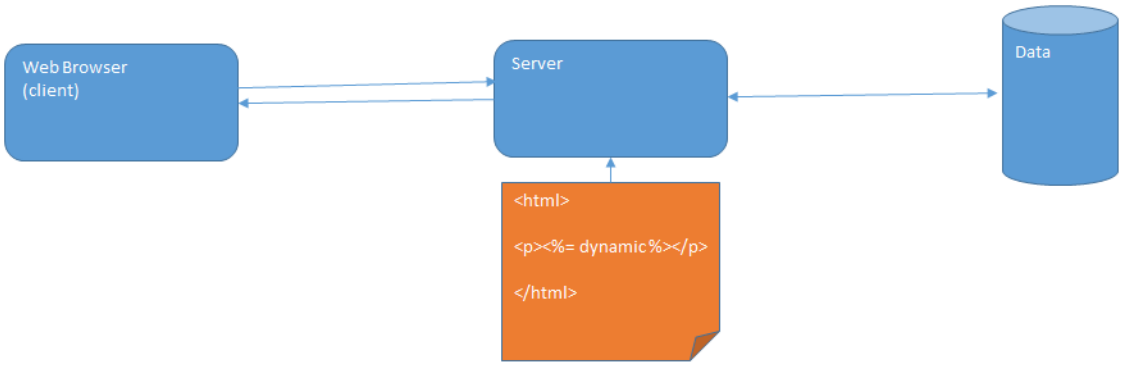
This may remind you of AJAX, an acronym for Asynchronous JavaScript and XML, which is a group of interrelated web development techniques used on the client-side to create asynchronous web applications, which become popular about ten years ago. AJAX is an enabling technology for creating SPAs and was initially used to progressively enhance web pages generated on the server. This allowed certain user interactions to not cause full page reloads. SPAs take this idea further, so now all user interactions including navigating between views/sections/pages in the application do not reload the page. To say it another way, the first time we did AJAX we did it very conservatively and sprinkled it on top of our existing applications but with SPAs we are trying to fully utilize the browser as a runtime environment for web applications.

Application Architecture

This section discusses how the high level architecture behind single-page applications (SPAs) works by contrasting it with traditional multi-page web applications.

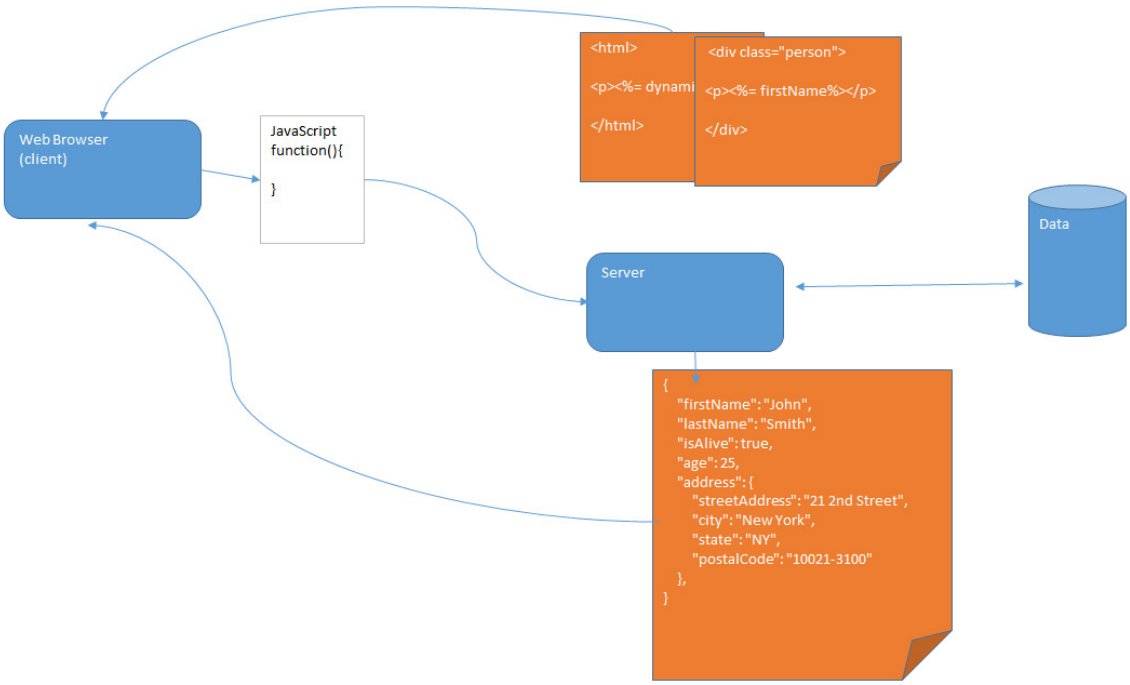
Server-side web applications

Multi-page web applications create their HTML on the server by combining templates (HTML) with dynamic data on the server and then send both down to the client together.



Client-side web applications

Single-page applications send the templates and dynamic data down to the client separately and assemble it into an HTML page on the client in the browser. With SPAs, the server's role is simplified to serve static templates and to separately serve the data required by those templates. As mentioned before, the key difference is the combining of templates and data on the client (using JavaScript, the universal language in web browsers).



Steve Sanderson, the creator of KnockoutJS one of the more popular JavaScript libraries that enables SPAs, put it best when describing the commonality in the explosion of Javascript libraries to facilitate SPAs:

"All the technologies follow from the view that serious JavaScript applications require proper data models and ability to do client-side rendering, not just server rendering plus some Ajax and jQuery code."

The quote below from the Throne of JS conference website summarizes the mindset behind SPA development:

It's no longer good enough to build web apps around full page loads and then progressively enhance them to behave more dynamically. Building apps which are fast, responsive and modern require you to completely rethink your approach.

Examples

Let's look at some real-world examples of single-page applications to help us better understand them.

Gmail

Gmail is now the most used web-based email client surpassing Yahoo! Mail, Hotmail, and AOL so even my mom now has an account. It has lots of great features that have contributed to its popularity but I would argue that the desktop feel of a native email client is the one of the most important.

Trello

Trello is flexible software to help you organize projects of many different kinds. It's free to try and is an excellent example of a more complicated single-page application.

Mint.com

The finance site has different modules of functionality including overview, transaction, budget, etc., but when you are navigating within one of these modules the page does not reload providing an extremely fluid experience to the user.

Facebook

The social network also has different modules of functionality including news feed, messages, photos, events, etc. , but once you are within one of these areas you generally stay within a single page. When I first started using it I liked it better than MySpace at the time, and it wasn't just the design, it was the fact that the page seldom reloaded giving me a more fluid experience.

Twitter

Twitter is another example, although it is admittedly a much simpler user interface than some of the previous web applications mentioned.

It's interesting to note that these sites are not only good examples of single-page applications that most people are familiar with but also, not coincidentally, the sites where I spend most of my time on the Internet.

JavaScript's Evolving Role

Given their architecture, single-page applications require a substantial part of the application to run entirely in JavaScript which has resulted in an ongoing explosion of libraries and frameworks to help developers build complex applications in what was previously a language used for user interface enhancements. The libraries and frameworks from the outside all look like yet another JavaScript library but once you understand the fundamental shift taking place in the way web applications are built from multi-page applications on the server to single-page on the client using JavaScript it becomes more apparent why this is happening. To be more specific there are numerous web development problems such as templating, data binding, routing, continuous integration, deployment etc. which have been solved on the server in PHP, Ruby on Rails, JSP (Java), ASP (.NET), Python, etc., but now need to solved on the client in the browser using JavaScript. Some of the JavaScript libraries are even addressing more general software development concerns such as object-oriented language

features, working with arrays and collections as well as modularity and organization of code. Without the use of these libraries and frameworks as building blocks this new breed of single-page applications can quickly become a mess of spaghetti code that is difficult to maintain or extend and prone to defects.

Summary

In this chapter we described what a single page application is and contrasted it with the more traditional multi-page application architecture. We also discussed the explosion of JavaScript libraries and frameworks and how this is partially due to the increase in the adoption of single-page application architectures to create a more "desktop like" experience for web users. In the next chapter, we'll dig deeper into the specific problems the JavaScript libraries are trying to solve.

CHAPTER 2

THE RISE OF JAVASCRIPT OPEN SOURCE

JavaScript Open Source

There are a wealth of open source JavaScript libraries to assist you in building the next generation of web applications. These libraries provide functionality we already have good library support for on the server-side (PHP, Ruby, Java, .NET). But now we have a need to do these same things on the client-side with JavaScript. In this chapter we'll discuss some of the more prominent JavaScript libraries at a high level to help you understand the overall landscape.

DOM Manipulation

DOM Manipulation is foundational and required in some form by all MVC frameworks. These libraries allow developers to query HTML elements from the DOM and change them in various ways while sheltering the developer from the numerous web browser inconsistencies. jQuery is the most well-known and de facto standard library that helps hide browser inconsistencies from the developer. In addition, there are some frameworks that only support DOM manipulation in modern browsers such as ZeptoJS (essentially dropping support for older versions of Internet Explorer). ZeptoJS is generally used for mobile development scenarios and is significantly smaller in size than jQuery.

Templating

Templates can be entire pages of HTML but more commonly are smaller fragments of HTML with data binding placeholder expressions included for dynamic data. Libraries that help with templating include Handlebars.js which is the most popular one. Handlebars.js is frequently used with Backbone.js and included as part of Ember (Handlebars can be replaced with Ember.js but you lose a lot of productivity and it's not recommended). Mustache.js is another popular templating engine. Underscore.js is a dependency of the Backbone framework and is a utility library with a micro-templating library as well as lots of functional

programming stuff. Dust.js is the rising star in this space being recently chosen by LinkedIn for use in their applications. Handlebars.js and Mustache.js are also frequently used in applications with just jQuery and AJAX (no JavaScript MVC) when developers realize they need templating on the client. jQuery was working on its own templating library but it has been deprecated and is not recommended.

MVC/MV*

The Model View Controller (MVC) pattern has been a clear winner as the most popular way of building web applications on the server. Other presentations patterns such as the Model View View Model (MVVM) pattern and the Model View Presenter (MVP) are commonly used as well. These prominent presentations patterns are collectively referred to as the MV* (Model View Star) or MVW (Model View Whatever) patterns which all have the goal of organizing the front end code in a web application. Numerous MV* frameworks with different goals and scope of features have been created in JavaScript to enable the user interface and its interactions to be handled on the client in the browser after shipping the data down via an API.

Framework or Library

The MV* libraries can be divided into two categories frameworks and libraries. Frameworks are larger in scope and tend to provide most if not all the functionality needed in one package. Other libraries are smaller in scope and tend to implement one or two of the features needed to facilitate building a rich client experience in JavaScript, for example just data binding.

AngularJS and EmberJS are clearly frameworks that try to provide a complete cohesive MVC architecture. BackboneJS is frequently compared to these frameworks but is really more of a user interface utility library providing a significant number of features without a strict adherence to the MVC design pattern or strong opinions about how an application should be built. The flexibility of the BackboneJS library is frequently viewed as strength and its breadth

of useful features cause it to frequently be compared to AngularJS and EmberJS. BackboneJS has been around significantly longer than these other frameworks so it is compared to them and can be thought of as a mature first-generation library that has inspired the second generation frameworks such as AngularJS and EmberJS.

Several other frameworks have had significant audiences and influence in the MV* area including CanJS, Batman, and SpineJS (very similar to BackboneJS).

Meteor

Meteor is also considered a framework but is broader in its scope stretching beyond MVC on the client to a full stack framework including NodeJS on the server and MongoDB as the default database. Meteor is not as mature as the others but shows great promise.

DurandalJS

DurandalJS is the other library worth mentioning when considering a more overarching framework. DurandalJS helps you with navigation, application life cycle, and view composition and facilitates a best of breed solution by being a common framework that other JavaScript libraries can be plugged into to create a more complete MVC architecture for an application. DurandalJS was written by Rob Eisenberg who previously created the Caliburn Micro UI Framework for WPF, Silverlight, Windows Phone 7, and WinRT/Metro. DurandalJS's initial adoption has been with developers with .NET experience and is commonly used with KnockoutJS since it originates in the same developer community. Recently, Rob Eisenberg announced he is now working on the AngularJS team on version 2.0 of the framework. He mentioned DurandalJS 2.x will be maintained but suggested a migration path.

MarionetteJS

MarionetteJS is similar to DurandalJS for BackboneJS applications and is a composite application library that includes pieces inspired by composite application architectures, event-driven architectures, and messaging architectures.

This book is about these MV* libraries and frameworks so we'll dive much deeper into them in other chapters. This section is just to help you understand the types of libraries out there and what specific problems each library is trying to solve.

Other JavaScript Libraries

Although I provided an overview of the MV* specific libraries and frameworks there are still a ton of JavaScript libraries you've probably heard about that don't fit under that umbrella. Let's review some of them quickly by category.

Data Persistence

BreezeJS

BreezeJS is a JavaScript library to help with data storage and persistence in MVC applications. Most other MVC frameworks just provide generic hooks and don't do a whole lot to help with data storage on the server. BreezeJS has features such as querying data on the client and server with .NET LINQ like syntax, caching data on the client and syncing it with the server in connected and disconnected scenarios. The library allows a developer to navigate from an aggregate root parent object to its child objects. BreezeJS can work with many server-side frameworks but is the most mature with a .NET back-end where its initial development was focused. In recent reports I've heard BreezeJS has developed excellent support for the MEAN stack as well (Mongo, Express, AngularJS, and Node).

Compilation

LESS and SAAS

LESS and SAAS are CSS preprocessor languages which means you can write code to generate your CSS files. Generating CSS dynamically allows for interesting features such as variables to easily change colors globally in your style sheet, mixins that enable you to compose CSS

styles out of previously defined styles, and hierarchical indented style definitions that remove repetition in your style sheet.

CoffeeScript

CoffeeScript is a programming language that compiles to JavaScript. The language adds syntactic sugar inspired by Ruby, Python and Haskell to enhance JavaScript's brevity and readability, adding additional features like list comprehension and pattern matching. CoffeeScript compiles predictably to JavaScript, and programs can be written with less code, typically 1/3 fewer lines (to be clear a lot of these lines are curly braces), with no effect on runtime performance.

CSS

Bootstrap

Bootstrap, formerly known as Twitter Bootstrap, is a standard set of HTML and CSS to style common elements including typography, forms, buttons, navigation and other interface components, as well as optional JavaScript extensions. Bootstrap has more followers on GitHub than jQuery so it is definitely worth evaluating.

Object-Oriented

JavaScript is a prototypical language and doesn't have what we traditionally think of as classes and interfaces although it does have constructor functions which act a lot like classes. Some libraries have added various objects and methods to help developers take a more object oriented approach to their design problems including *Mootools*, *PrototypeJS* and the "Model" parts of some MV* frameworks most notably Backbone and Spine.

Functional Programming

UnderscoreJS

UnderscoreJS provides a lot of the functional programming support like you've seen in other libraries including Prototype.js and languages such as Ruby, but without extending any of the built-in JavaScript objects. Underscore provides about 80 functions that support both the usual functional suspects: map, select, invoke -- as well as more specialized helpers: function binding, JavaScript templating, deep equality testing, and so on. It delegates to built-in functions, if present, so modern browsers will use their native implementations of forEach, map, reduce, filter, every, some and indexOf if available.

Origins of Underscore

Underscore came about when Jeremy Ashkenas, the creator of Backbone, extracted Underscore out of the original versions of Backbone so that the useful utility stuff could be used by people who were not using Backbone.

Modularity

RequireJS

RequireJS is an asynchronous file and module loader which really means that instead of having 30 script tags ordered very specifically at the top of every page you'll have one script which has a RequireJS configuration in JavaScript that understands the dependency between your various scripts because they have all been wrapped into distinct modules. It will load these modules only when needed which will speed the performance and increase the maintainability of your application. RequireJS follows the Asynchronous Module Definition (AMD) standard but focuses on doing this asynchronously in the browser while other AMD libraries focus on organizing server-side JavaScript code. In addition to not having to worry about script order RequireJS helps clear up the global namespace because when scripts are packaged into modules they are given a namespace to keep that code separate.

Unit Testing

TDD

QUnit

QUnit is a powerful, easy-to-use JavaScript unit testing framework. It's used by the jQuery, jQuery UI and jQuery Mobile projects. It is similar to JUnit for Java or NUnit or MSTest for .NET.

BDD

Jasmine

Jasmine is a behavior-driven development framework for testing JavaScript code. It uses a different syntax for writing the unit tests to make them more understandable and closer to requirements. The Jasmine syntax is very similar to RSpec for Ruby developers. The AngularJS project uses Jasmine as its default test runner.

Mocha

Mocha is more of a framework to build your own unit testing framework allowing you to plug in your own assertion syntax and testing syntax (BDD or TDD style). Chai is a popular assertion syntax library commonly used with Mocha so you'll hear them mentioned together frequently.

Mocking

Sinon

Sinon.js is a popular library for creating spies, stubs, and mocks in JavaScript and can be used with any of the aforementioned libraries.

Task Runner

Grunt

Grunt is a command-line task runner which has a great ecosystem of plugins for doing repetitive tasks like minification, compilation, unit testing, linting, etc, of JavaScript files.

Gulp

Gulp is similar to Grunt but is stream based instead of file based for its input/output (IO) so it is significantly faster. The community is also making an effort to keep a clean plug-in repository but at the time of writing they do not have as many plugins as the more mature Grunt.

JavaScript on the Server-Side

Node.js

Node.js is primarily an environment enabling developers to use JavaScript to program server-side as well as client-side. It runs on Chrome's JavaScript runtime (the Google V8 engine) and has a novel asynchronous non-blocking I/O model which allows it to be extremely performant

and scalable compared to traditional server-side web servers and programming runtime environments. Since it has a runtime outside the browser it is also the foundational technology used under the hood in most of the JavaScript build and deployment tools mentioned above.

Package Manager

Bower

Bower is a package manager for client-side code. Differentiating between Bower and nodes.js's package manager, npm, can be a source of confusion for developers. Essentially Bower is for front-end JavaScript code that will run in your browser like jQuery, Backbone, Ember, etc. while npm is for server-side code that will run inside node on the server like the "request" or "express" packages that add functionality to your web server.

Scaffolding, Command-line IDE

Conventions

You'll notice that these JavaScript applications don't really have an integrated development environment (IDE) and are generally built using your favorite text editor and the command-line. This can be quite liberating but the blank slate can often make it hard to get started quickly because of the lack of conventions commonly provided by project templates. Two tools that are trying to solve this problem but have different approaches are *Yeoman and Lineman*. These approaches could be a chapter themselves so I will just point them out here to increase awareness and point out another area that is mature on the server but experiencing rapid innovation on the client-side.

CHAPTER 3

SHOULD I BE USING A JAVASCRIPT FRAMEWORK?

Questions

As people became aware that I was working on this book I've been asked one question repeatedly:

Should I be using a JavaScript Framework for my project?

Here are some of the specific questions I have been asked.

> *"I'm in here updating Rails and Bootstrap and all my gems, should I stop and consider a JavaScript MVC framework? I seem to be getting along fine without one, but am I missing out? What are the top N reasons I would want something like Backbone or Ember in my project?" -Rails Developer*

> *"We have a complex web app that is in ASP.NET Web Forms. The existing code uses the code-behind page to do initial loading of content. However, after that, it's updated via AJAX postbacks using jQuery. I was wondering if a front-end framework might be better for us. I would think existing Web Forms apps are a common reason people might want to look into another option. Am I thinking on the right track or not? Would something like Angular help me?"*
> *- .NET Developer*

"My company is beginning development on a new web application that is strategic and will drive much of the future revenue. Bookmarking and search engine optimization are important in my application. Should I build it as a single-page application? Which of the frameworks should I use?" - PHP Developer

"I have this Silverlight application and I've been considering rewriting it because I'm concerned about future support for the technology from Microsoft. Would it make sense for me to rewrite this application using one of the JavaScript frameworks?
- .NET Developer

These questions are unique and complex and each answer I've given has been different but I've managed to boil the answers down to a few more specific questions to help inform all these use cases.

Why build a single-page application and not a traditional server-side web application plus jQuery? What are the business reasons? What are the technical reasons?

What are the telltale characteristics of a project that would benefit from being a single-page application?

What are the pros and cons of various technical architectures that could be chosen to deliver a single-page application user experience? (i.e. OK, I'm sold on a single-page application for my project now why JavaScript MV)*

To help answer these questions we will establish the choices or alternative architectures being considered in the next section.

Architectures

Let's quickly describe the architectures developers are considering for web applications so we can compare and contrast them.

Client-side MV*

With client-side Model-View-Whatever (MV*) frameworks, the web browser (client) requests the templates (static HTML) and data (usually JSON) from the server and assembles them in the browser using JavaScript.

Server-side plus jQuery

In a server-side web application plus jQuery the server generates the HTML and sends it to the client and the markup is modified using jQuery selectors to grab DOM elements and update their contents with either updated HTML or data from the server.

Technical Reasons

Now that we have a mental model of the common architectures let's answer the first question.

Why build a client-side single-page application and not a traditional server-side web application plus jQuery?

Just to reiterate what we are comparing is server-side rendering of HTML progressively enhanced with jQuery code which does callbacks to the server and re-renders sections of the page or adds interactivity versus a single-page application built using a JavaScript MV*

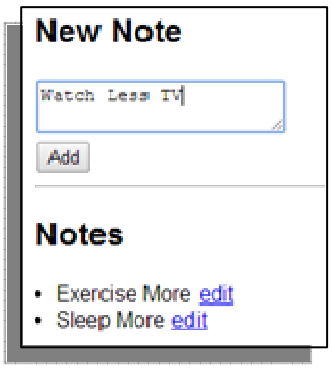
framework or library.

Organization and Maintainability

One of the most frequently cited reasons to choose a client-side single-page architecture is improved organization and maintainability of your JavaScript code but that is vague, so I'm going to give some specific examples using a sample application.

Notes Example Application

I'll use a simple notes application that allows the user to enter a note in an input (textbox) and when the user clicks the add button the note gets added to a list of notes at the bottom of the page. When the add button is clicked the note is added to the list on the user interface but also a fake call to a web API is made to simulate persisting the note in a database and a unique ID for the note is returned. Lastly, there is an edit link next to each item on the list which alerts the unique ID of the persisted note to the user. I've tried to create the simplest example possible while still being able to show some of the differences in these architectures.



jQuery Notes Example

First, let's look at the application written with jQuery (yes, I could improve this code but I believe this reflects how this task generally gets done using jQuery).

```
//html
<div id="new-note">
    <h2>New Note</h2>

    <form action="">
        <textarea></textarea>
        <br>
        <input type="submit" value="Add" />
    </form>
</div>
<hr>
<div id="notes">
    <h2>Notes</h2>

    <ul></ul>
</div>

// javascript
$(document).ready(function () {
    $('#new-note form').submit(function (e) {
        e.preventDefault();

        var request = $.ajax({
            type: "post",
            url: "/echo/json/",
            data: {
                json: JSON.stringify({
                    text: $('#new-note').find('textarea').val(),
                    id: 1000
                })
            },
            dataType: 'json'
        });

        request.done(function (note) {
            $('#notes ul').append('<li data-id=' + note.id + '>' + note.text
            + '<a class="edit" href="#">edit</a></li>');
            $('#new-note').find('textarea').val('');
        });
```

```
    });

    $("#notes").on("click", ".edit", function (e) {
        e.preventDefault();
        var id = $(this).parents().first().attr("data-id");
        alert("You are editing the record with id: " + id);
    });
});

// here is the JSFiddle for the jQuery notes app.
```

Backbone Notes Example

And here is the same application written using jQuery and Backbone.

```
// html
<div id="new-note">
    <h2>New Note</h2>

    <form action="">
        <textarea></textarea>
        <br>
        <input type="submit" value="Add" />
    </form>
</div>
<hr>
<div id="notes">
    <h2>Notes</h2>

    <ul></ul>
</div>
<script type="text/template" id="note-template">
    <%= text %>
    <a class="edit" href="#">edit</a>
</script>

// javascript
var Note = Backbone.Model.extend({
    url: '/echo/json/'
});

var Notes = Backbone.Collection.extend({
    model: Note
});

var NewNoteView = Backbone.View.extend({
    events: {
        'submit form': 'addNote'
    },

    initialize: function () {
        this.collection.on('add', this.clearInput, this);
    },

    addNote: function (e) {
        e.preventDefault();
        this.collection.create({
            text: this.$('textarea').val(),
```

```
            id: Math.floor((Math.random()*100)+1)
        });
    },

    clearInput: function () {
        this.$('textarea').val('');
    }
});

var NotesView = Backbone.View.extend({
    initialize: function () {
        this.collection.on('add', this.appendNote, this);
    },
    appendNote: function (note) {
        var noteView = new NoteView({model:note});
        this.$('ul').append(noteView.render().el);
    }
});

var NoteView = Backbone.View.extend({
    tagName: "li",
    template: _.template($("#note-template").html()),
    events: {
        'click .edit': 'editing'
    },
    render:function(){
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    },
    editing: function (e) {
        e.preventDefault();
        alert("You are editing the record with id: " + this.model.id);
    }
});

$(document).ready(function () {
    var notes = new Notes();
    new NewNoteView({
        el: $('#new-note'),
        collection: notes
    });
    new NotesView({
        el: $('#notes'),
        collection: notes
    });
});

// Here is the JSFiddle for the Backbone version of the notes app.
```

Problems and Solutions

The next section will discuss problems in the jQuery version and how features in the JavaScript MV* version provide a solution to each problem. These issues are admittedly not a big deal in this small application but imagine these problems in a bigger, more complex code base.

Problem: HTML in JavaScript

Every time a note is added to the list on the user interface a new item (
) is added to the unordered list (<ul>). This item template is hard-coded in JavaScript in the jQuery version below.

```
request.done(function (note) {
    $('#notes ul').append(' <li data-id=' + note.id + '>' + note.text + '<a
      class="edit" href="#">edit</a></li>');
    $('#new-note').find('textarea').val('');
});
```

Solution: Templates and Data binding

The solution is to move the item template out of the JavaScript into an HTML view template as we did in the Backbone version:

```
<script type="text/template" id="note-template">
    <%= text %>
    <a class="edit" href="#">edit</a>
</script>

appendNote: function (note) {
    var noteView = new NoteView({model:note});
    this.$('ul').append(noteView.render().el);
}
```

This isn't too big a deal, because the template is so simple but templates quickly become more complex in my experience. Let's imagine a few ways this template might become more

complex:

- List items may need CSS classes but which classes the item gets depends on whether it's an item just added, or an item being edited etc...
- List items may have another list nested inside of them, for example a series of posts or questions with comments. The nested list of comments may be complex enough to need its own template
- List items may each have a form with several inputs and buttons

When this complexity comes in it becomes a mess to manage these HTML fragments constantly escaping back and forth between data and markup and either loading these fragments into jQuery wrapped DOM elements to modify them easily or doing string replacement with regular expressions. Templates solve this problem and make the code much easier to follow and less complex.

Problem: HTML Templates Duplicated on Client and Server

One of the commonly referenced S-O-L-I-D principles is don't repeat yourself (D-R-Y) but when you render HTML on the server and then re-render that same HTML or a fragment of that HTML on the client in JavaScript (this happens when you return data/JSON from the web API and need to copy it into the template on the client) you run into the need to have a template for that markup on both the server and the client in two different templating languages.

In the jQuery example the notes were originally rendered from the server so the list item has a template on the server which probably looks something like this:

```
<ul>
    <% foreach (note in notes)%>
    <li data-id='<% = note.id %>'> <% = note.text %>  <a class="edit"
      href="#">edit</a></li>
    <% } %>
</ul>
```

Issues can be introduced whenever you change the template on the server because it will also need to be changed in the JavaScript code or template.

For example, let's say we add a class to indent the list items on the server:

```
<ul>
    <% foreach (note in notes)%>
    <li class="indent" data-id='<% = note.id %>'> <% = note.text %>  <a
      class="edit" href="#">edit</a></li>
    <% } %>
</ul>
```

We could easily forget to do this in the JavaScript code or the client-side template (if you are using a template library like handlebars). Even more likely than forgetting is that another developer or a designer may not know there are two copies of that template (one for the client and one for the server).

Solution: Thin-Server Architecture

"Thin-server architecture" is the idea that you have a thin RESTful data API on the server that serves multiple rich or thick clients. For example, a single-page JavaScript application and a native mobile application get the same data from an API. Thin server architecture solves the duplicate template problem by often needing only one template which is interpreted on the client in JavaScript. All of the JavaScript MV* frameworks assume the use of a "thin-server architecture."

Problem: Views and Model Data are too tightly coupled

In a server-side plus jQuery architecture, the "application state" is frequently stored in the DOM in the ID attribute, class or a custom attribute of an element. The most common example of this is unique IDs from a database being stored as attributes in the DOM so they can be referenced later to pull back the other "application state" or model data. This tight coupling between the user interface and data becomes a problem when a designer or developer changes a CSS class name or ID on an HTML element and breaks the application. I've learned over the years that having more than one copy of data in an application frequently results in one of the copies getting out of sync and causing bugs.

In the jQuery example you see a unique database ID being stored in the DOM:

```
$('#notes').append('<li data-id=' + note.id + '>' + note.text + '<a
class="edit" href="#">edit</a></li>');
```

The unique id is later used when editing the note:

```
var id = $(this).parents().first().attr("data-id");
```

Solution: Models and Data binding

With client-side MVC frameworks, that "application state" is managed in JavaScript objects in memory similar to how a traditional desktop application manages its "application state". The data-id attribute isn't even needed in the Backbone example.

```
<script type="text/template" id="note-template">
    <%= text %>
    <a class="edit" href="#">edit</a>
</script>
```

This is possible because the note view is an object that manages a specific list item <li> on the page and has all the model data (a note object) associated with that DOM element in the web browser's memory (not just the id). Backbone views all have a property named "model" which represents the view model for that HTML template or view.

```
var NoteView = Backbone.View.extend({
    tagName: "li",
    template: _.template($("#note-template").html()),
    events: {
        'click .edit': 'editing'
    },
    render: function () {
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    },
    editing: function (e) {
        e.preventDefault();
        alert("You are editing the record with id: " + this.model.id);
    }
});
```

Problem: Organization and Maintenance

The jQuery example code is not organized because all the code is contained in a few functions and those functions are not organized to have only one reason to change--i.e. a single responsibility. The view and model data logic are inter-tangled. The code used to render the entire page or view (HTML) and all the events triggered by that large view as well as all code to manage the data needed for the view including how to store it and retrieve it is in one place. That is a lot of responsibility for those couple functions.

```
var request = $.ajax({    //data code
    type: "post",
    url: "/echo/json/",
    data: {
        json: JSON.stringify({
            text: $('#new-note').find('textarea').val(), //View code
            id: 1000
        })
    },
    dataType: 'json'
});
```

In object-oriented programming, the single responsibility principle states that every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility. A responsibility

is defined as a reason to change. The reason it is important to keep an object focused is because changes break more code and can be difficult to fix if they are not isolated to a certain part of a system.

Maybe you haven't mastered the S-O-L-I-D principles (the S is single responsibility mentioned earlier) but just think about how you would write the same code on the server using a server-side MVC framework. You would have a view or partial with your html, a model to represent your data, and a controller to react to user interface events and bind your model together with the view. You wouldn't put all the code in one render method (which is basically what the jQuery example does). *These client-side frameworks are simply bringing the same type of code organization you do by habit on the server to your JavaScript code on the client.*

Solution: Views (in Backbone) or Controllers, Models, Events

JavaScript is a class-less object-oriented language and unlike most modern object-oriented languages used on the server which include familiar constructs for organization such as namespaces/packages/ modules, private scope, and reuse through classes. JavaScript can be written to include all these organizational code features but these JavaScript libraries and frameworks make it easier to fall into the pit of success and keep your code modular and organized.

```
var Note = Backbone.Model.extend({
    url: '/note/'
});

var NewNoteView = Backbone.View.extend({
    events: {
        'submit form': 'addNote'
    },

    initialize: function () {
        this.collection.on('add', this.clearInput, this);
    },

    addNote: function (e) {
        e.preventDefault();
```

```
        this.collection.create({
            text: this.$('textarea').val(),
            id: Math.floor((Math.random() * 100) + 1)
        });
    },

    clearInput: function () {
        this.$('textarea').val('');
    }
});

$(document).ready(function () {
    var notes = new Notes();
    new NewNoteView({
        el: $('#new-note'),
        collection: notes
    });
    //code omitted
});
```

Notice how every line of code has a logical place to live and each object has a single-responsibility. Another benefit of this approach is models can be reused with other views.

Events and Observable

One feature that makes it easier for these views and models to live separately is that they can send events back and forth. The most common example of this is the observer pattern where change tracking is enabled on a model and change events such as "add" or "change" fire and the views can listen for these and take appropriate action such as re-rendering.

Problem: Back Button Broken, Bookmarking doesn't work

Another challenge is state management. In a traditional server-side web application "state" is commonly maintained in the URL of the web browser. When you reload parts of the HTML on a page but don't update the URL of the web browser the back button may not work in all the ways useful to the end users without careful consideration.

Let's say you want to change from a list view to a list item details view you want the url to change so that users can bookmark and send each other direct links to the list item details. If

you just changed the content of a main div with jQuery then sent the current URL to someone in an email it would open to the original list instead of the details. You could use a routing library like history.js to keep the back button working with server-side plus jQuery but it's common to not find these issues until a tester or user reports the issue.

Solution: Routing

The solution to the list/details scenario is to introduce the backbone router into the existing example. Routing as described earlier allows you to listen for changes to the URL and then run client-side JavaScript code to load a new or additional template or view. In this example, the router detects the URL change to "#/details/:id" and then runs code to load the details (which aren't too exciting in this case because it's just the same note text you already can see in the list but it's easy to imagine a more complicated note model object with more details.

```
<div id="new-note">
    <h2>New Note</h2>

    <form action="">
        <textarea></textarea>
        <br>
        <input type="submit" value="Add" />
    </form>
</div>
<hr>
<div id="notes">
    <h2>Notes</h2>

    <ul></ul>
    <div id="note-details-container">
    </div>
</div>
<script type="text/template" id="note-template">
    <%= text %>
    <a class="edit" href="#">edit</a>
    <a class="details" href="#/details/<%=id%>">details</a>
</script>
<script type="text/template" id="note-details-template">
    Here are the note details:
    <%= text %>
    <br />
```

```
    (which is just the note text again)
</script>

//javascript
var Note = Backbone.Model.extend({
    url: '/echo/json/'
});

var Notes = Backbone.Collection.extend({
    model: Note
});

var NewNoteView = Backbone.View.extend({
    events: {
        'submit form': 'addNote'
    },

    initialize: function () {
        this.collection.on('add', this.clearInput, this);
    },

    addNote: function (e) {
        e.preventDefault();
        this.collection.create({
            text: this.$('textarea').val(),
            id: Math.floor((Math.random()*100)+1)
        });
    },

    clearInput: function () {
        this.$('textarea').val('');
    }
});

var NotesView = Backbone.View.extend({
    initialize: function () {
        this.collection.on('add', this.appendNote, this);
    },
    appendNote: function (note) {
        var noteView = new NoteView({model:note});
        this.$('ul').append(noteView.render().el);
    }
});

var NoteView = Backbone.View.extend({
    tagName: "li",
    template: _.template($("#note-template").html()),
    events: {
        'click .edit': 'editing'
```

```
    },
    render:function(){
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    },
    editing: function (e) {
        e.preventDefault();
        alert("You are editing the record with id: " + this.model.id);
    }
});

var NoteDetailsView = Backbone.View.extend({
    //tagName: "div",
    template: _.template($("#note-details-template").html()),
    initialize:function(){
        this.render();
    },
    render:function(){
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    }
});

var AppRouter = Backbone.Router.extend({
    NotesCollection :  new Notes(),
    routes: {
        '': 'listRoute',
        'list':'listRoute',
        'details/:id': 'detailsRoute'
    },
    listRoute: function () {
        new NewNoteView({
            el: $('#new-note'),
            collection: this.NotesCollection
        });
        new NotesView({
            el: $('#notes'),
            collection: this.NotesCollection
        });
    },
    detailsRoute: function (id) {
        new NewNoteView({
            el: $('#new-note'),
            collection: this.NotesCollection
        });
        new NotesView({
            el: $('#notes'),
            collection: this.NotesCollection
        });
```

```
        new NoteDetailsView({
            el: $('#note-details-container'),
            model: this.NotesCollection.get(id)
        });
    }
});

var appRouter = new AppRouter();
Backbone.history.start();

// JSFiddle link
```

But it's more code?

You might also notice that there is more code but it's in tiers or layers and objects have a single responsibility.

By analogy, if you took all the mail you got for a year and put it all in one box versus organizing it by bills, junk-mail, bank statements, insurance statements would you know where to look for your bills? I'm not saying organizing your mail is a good idea but the more time you spend trying to find things in your mail the more important it is to be able to find things quickly. The more mail you get the more important it is to keep it organized if you need to find it. The more important your mail, for example if you get a lot of checks in your mail, the more important it is to keep it organized.

Architecture

Another rationale for single-page applications centers around architecture and the fact that one resource-based API can be written to provide data to multiple rich client applications. More specifically, you could have a web application that relies on a web service to get data to display via AJAX calls as well as an iPhone application and an Android application that get their data from the same central API. This does present challenges for code reuse of the application code in JavaScript as in most mobile architectures but the data API can serve

multiple purposes.

Summary

Understanding these problems and the solutions provided by the frameworks can help you clearly articulate why bringing one of these frameworks into your project would be helpful.

Business Reasons

Being great is a Strategic Advantage

Being great is a strategic advantage for a business is an obvious statement but an important one. Sit your stakeholders down and have them use an older generation web-based email such as juno.com and then have them use Gmail. Gmail was not the first free web-based email. How did it beat out Juno and Hotmail? Gmail provided a better user experience and innovated on features in my opinion. A more native, slick application providing a strategic advantage has happened numerous times in the past. Remember MySpace users flocking to Facebook despite everyone being on MySpace at the time? Do you like the forums you use on the Internet most if which use phpBB? Have you tried Discourse? You can see this trend outside web applications with the PC versus Mac debate. Macs are perceived as luxury goods and that is a strategic advantage for them. Taking this example beyond technology, would you prefer a Toyota Camry or Lexus? The Lexus is a luxury good and enjoys a strategic advantage. I believe single-page applications are currently a luxury good and can be a significant strategic advantage if done properly. It's also worth noting that the things that differentiate a Lexus will become standard in a Camry in the next 5-10 years if not sooner (think power windows, keyless entry). Given this, I would predict that many, even most web applications might be written as single-page applications with heavy JavaScript usage in the next 5-10 years and it won't be considered a big decision but just the way things are done.

Many corporations now support iPhones instead of requiring Blackberry usage because users demanded it. Will users develop a new set of minimum expectations on how a web application should feel or work based on using Facebook, LinkedIn, Gmail, Twitter, Mint, and Trello? I think the answer is yes and they will evaluate the web applications you build based on these expectations. And being great will be a strategic advantage.

What it means to be great

So now you're sold on being great but what does it mean to be great in a web application?

Native feel

There are a lot of factors but the most important is to provide a native feel akin to a desktop application with smooth transitions between pages and sections of the application.

Speed

Web applications like Google search, Facebook, and Stackoverflow that treat speed as an important feature seem to succeed because speed is directly correlated to their business value. Marissa Mayer Yahoo CEO and former Google executive gave a famous talk at the Web 2.0 Summit on the positive impact of speed on Google Search, Google Maps, and Gmail. Looking back over Marissa Mayer's comments at Web 2.0 a few quotes stood out to me: "If you have each transaction take less time, you have expert users more satisfied. You want lots of small and fast interactions if speed is important," she added. "Lots of small and fast interactions" sounds like someone describing a single-page application.

Benefits of being great

Increased revenue

When your pages load faster you get increased user engagement/utilization and/or increased page views resulting in increased revenue. This increase can be direct for example if advertising is one of your major sources of revenue or indirect if the internal users of the application can be more productive or effective when enabled by "great" tools.

Direct

Examples include Google AdWords revenue goes up because utilization goes up of Google Search, Google Maps, Gmail, etc... Or more Facebook ads are seen because Facebook is more responsive. Or Amazon's revenue increases when their product pages load faster or their checkout is quicker or easier for end users.

Indirect

We've all seen applications that are not responsive but people adapt and learn to do their job around the system. Imagine a user kicking off a search in an application that takes a while and then surfing the Internet for 15 minutes before getting back to the original task. Consider how costly mistakes made by people using the system are and whether the system could decrease these mistakes. How does the web application affect your end customer's opinion of your company? For example how often have you called a customer service representative who commented that their system is slow and they are just waiting for it to load -- please give me a minute.

Innovation

Single-page application architectures allow specialization on the team that is innovating new product designs. For example, some developers with more expertise in JavaScript, HTML, and CSS can own the front-end and develop it by collaborating with graphic design and user experience specialists and later have server-side development resources model the data storage and services after the design has emerged and solidified. This allows for a much more agile, rapid prototyping life cycle and for team members to play to their strengths. For the business this can mean an increase in the pace of innovation or the effort taken to realize the vision.

Lower expenses

The better the user experience, the lower the training costs for the application. In a growing company, these training costs can be substantial. Marketing costs can also be lower and correlated to how much easier it is to demonstrate the value of the product and the viral nature of an easy to use product. Lower software maintenance costs may also result as client-side code becomes more organized.

Checklists

Now that we've covered some business and technical reasons why you might want to build a single-page application and not a traditional server-side web application let's get more specific and actionable and consider the question:

What are the telltale characteristics of a project that would benefit from being a single-page application?

Technical Checklist

What are the some specific signs your code could benefit from a JavaScript MV* library or framework.

- Duplicating HTML view templates on the client and on server
- Building HTML view templates by concatenating strings in JavaScript (even small fragments)
- Repeating the same jQuery selector in 3 or more places on the same page
- Storing state (such as database IDs) in the HTML (DOM) so you can restore the rest of an object's state when needed
- Nested callback functions are becoming difficult to maintain because of nesting and keep you from reusing the code inside the callback functions
- Functions that are too long and that do too much (more than one responsibility)

This commonly happens because the developer is unsure of variable scope in JavaScript. For example a function that does validation, and makes a data access call via AJAX to the server, and then updates the page for success and error conditions.

- Back button is broken, but your app is REALLY cool.
- Repeating presentation logic on client (in JavaScript) and on server (PHP, Rails, ASP, JSP, etc...)
- JavaScript code needs more structure and organization. The application's code declares functions in a file but not organized inside an object. The objects are not organized in namespaces or modules as is common is server-side languages. In JavaScript this is what is called "global namespace" pollution. In single-page applications it means you are hanging all your code off of the browser's window object. Open a page in your application and run this command in the JavaScript console of Google Chrome to visualize this problem: keys(window) Note: there will be 35 to 40 objects there but all your code should be under a few objects

- Complex operations on collections such as sorting and filtering are done on the server because it's easier even though you have all the data on the client

Business Checklist

What are some specific business reasons your project could benefit from a JavaScript MV* library or framework?

- Is it an application? Do the application requirements ask for an application or a more static/brochure ware website (with SEO as a priority)? Frequently, the answer can be a hybrid, for example an e-commerce site may need product pages with great SEO but also want a slick checkout experience for users. Maybe the site also has a rewards program that is more of an application. Single-page application functionality can become modules of richness in a bigger application. Another hybrid example is the popular Project Management software Basecamp by 37 signals uses traditional server-side rendering for the project and todo lists but uses backbone.js for the calendar module of the application.
- Is the application used by lots of users? To get a return on the investment from the application development costs consideration should be given to how many users will use the application. Consider future growth of users (new employees) as well as current users and factor in the training costs associated with new users.
- Do the users spend a lot of time in the application? The more time users spend the easier it is to justify your investment in the application by quickly recouping savings in increased productivity.

Do you need a custom hand-crafted chair or do you just need a plastic chair from Walmart? This gets back to the luxury good versus commodity arguments I made earlier, if you need "great", if you need a "strategic advantage" then consider a single-page application. If you need a plastic chair then perhaps building a traditional web application will suffice.

Server-side HTML via AJAX

So after looking at the business reasons for a single-page application you are convinced that an application with a more native feel would be ideal for your project but you're not convinced a JavaScript MV* framework is the best way to achieve this result.

OK, I'm sold on a single-page application for my project now why JavaScript MV?*

The decision on single-page application architectures is not as simple as "Server-Side Web Application Plus jQuery" or "JavaScript MV* Framework". There is some middle ground on the continuum between these choices. I've already addressed architecture, code maintenance and organization concerns with Server-Side Web Application plus jQuery approaches but there is one other architecture that is worth consideration when in need of a single-page application experience.

PJAX as an Architecture

The term PJAX is just an amalgamation of the browser feature "push-state" (P) and the better-know Asynchronous JavaScript and XML (AJAX). In a PJAX architected application, the initial HTML page is rendered to the browser and subsequent AJAX requests are made to get partial pages (HTML fragments not JSON data) from the server and replace sections of the DOM with them.

PJAX is a possible architecture pattern but can be confusing because there is a popular jQuery plugin by the same name that implements the pattern.

PJAX as a Plugin

One of the more popular ways to implement a PJAX architecture is the PJAX Plugin for jQuery. Essentially it is a wrapper around jQuery's "$.ajax()" functionality. It uses the new "pushState" and "replaceState" methods to manipulate the browser's history.

Basecamp

Basecamp project management software is known for being fast "without killing our development joy by moving everything client-side" in the words of David Heinemeier Hansson. Basecamp uses an internal PJAX architecture implementation and caching to get the performance they need. Their architecture is discussed in more detail in this blog post.

Github

Github has been reported as using a PJAX architecture to implement their file tree browser.

Pros

The simplicity of this approach coupled with the maturity of the tools (IDEs, languages, build environments) on the server make a PJAX architecture worth considering. In addition, it is generally easier to support a wider array of older browsers with this architecture. Another sign you may want to consider this approach is when your requirements are to build more of a "site" with sheets or documents of content than a long-lived application. For example if I was building a Web Content Management System (CMS) I would consider it but if the requirements are more for forum software with tons of user interaction then I'd lean towards JavaScript MV*.

Cons

Since PJAX is more of a pattern than a specific library or framework there is a tendency to "underdo it" and come up with solutions that work but are robust enough to work as requirements evolve and change. Given that the solutions are often home grown it also becomes difficult to recruit developers who are familiar with and can hit the ground running with the proprietary solutions that are developed.

Summary

In this chapter we've closely examined the question of "Should I be using a JavaScript Framework for my project" from both a technical and a business perspective. Using what you've learned you should be empowered to have an informed discussion and confidently decide whether these technologies are needed on your project.

CHAPTER 4

HOW TO LEARN FRAMEWORKS QUICKLY

Learn Quickly

The key to quickly learning JavaScript MV* Frameworks is to break them down into a series of features. The main features of an MV* application are routing, data binding, templates/views, models, and data storage. In this post I'll describe these main features and show code examples from two or three frameworks for each feature. You will begin to concretely understand what these frameworks are trying to help you accomplish and realize they are more alike than they are different. In fact, it becomes apparent that most of the frameworks borrow heavily from the successes of the others.

"I tend to see the similarities in people and not the differences." -- Isabel Allende

Don't be too concerned about understanding every line of code. I'll explain the details of each framework in the example application chapters. For now, try to appreciate how similar they are and the problems they can solve for your project.

Routing

Routing, at a minimum maps your URLs to actions, but sometimes goes as far as implementing a full "state machine" design pattern for managing state transitions within a view.

If you've ever used the router in a server-side MVC framework such as Rails, CodeIgniter, CakePHP, ASP.NET MVC, Spring MVC, JSF, STRUTS, Grails, etc... then you can just think of the JavaScript MV* routers as the same thing but running on the client in JavaScript and routing to other JavaScript objects instead of running on the server and routing to server-side code

written in PHP, Ruby, Java, C#, etc..

So you may be wondering how can this work and will this work on older browsers? It works by default by using everything after the hash tag in a URL as the route. However, if HTML push-state support is configured (with one line of code in most frameworks) then URLs without hashes that match the routes will be intercepted on the client and run JavaScript as well.

Push-state is NOT supported by default because URLs directly requested via bookmarks or sent in an email will work for the end user but search-engine crawlers do not have great support for JavaScript, i.e., they may not run the JavaScript code and subsequently won't see the correct content on the page. The generally proposed solution to this is to run PhantomJS or another lightweight browser on the server and have it load the pages and JavaScript and return the generated markup and JavaScript. This takes some work to setup, hence the default to the hash URLs that all browsers support.

Enough details, let's see some code.

Backbone Routing Example

Here is a simple example of routing in Backbone.js

```
//html
<div id="navigation">
    <a href="#/home">Home</a>
    <a href="#/about">About</a>
</div>
<div id="content">
</div>

//javascript
var HomeView = Backbone.View.extend({
    template: '<h1>Home</h1>',
    initialize: function () {
        this.render();
    },
    render: function () {
        this.$el.html(this.template);
    }
});


var AboutView = Backbone.View.extend({
    template: '<h1>About</h1>',
    initialize: function () {
        this.render();
    },
    render: function () {
        this.$el.html(this.template);
    }
});

var AppRouter = Backbone.Router.extend({
    routes: {
        '': 'homeRoute',
        'home': 'homeRoute',
        'about': 'aboutRoute',
    },
    homeRoute: function () {
        var homeView = new HomeView();
        $("#content").html(homeView.el);
    },
    aboutRoute: function () {
        var aboutView = new AboutView();
        $("#content").html(aboutView.el);
```

```
    }
});

var appRouter = new AppRouter();
Backbone.history.start();

//JSFiddle
```

In particular notice the AppRouter object. Routes are mapped to functions. The functions simply create a view object which manages a DOM fragment and add it to the page when the URL changes. The Backbone.history.start() tells Backbone to start listening for URL changes.

AngularJS Routing Example

Here is a simple example of routing in AngularJS.

```
//html
<script type="text/ng-template" id="embedded.home.html">
    <h1> Home </h1>
</script>
<script type="text/ng-template" id="embedded.about.html">
    <h1> About </h1>
</script>
<div>
    <div id="navigation">
        <a href="#/home">Home</a>
        <a href="#/about">About</a>
    </div>

    <div ng-view></div>
</div>

//javascript
var routingExample = angular.module('FunnyAnt.Examples.Routing', []);
routingExample.controller('HomeController', function ($scope) {});
routingExample.controller('AboutController', function ($scope) {});

routingExample.config(function ($routeProvider) {
    $routeProvider.
    when('/home', {
        templateUrl: 'embedded.home.html',
        controller: 'HomeController'
    }).
    when('/about', {
        templateUrl: 'embedded.about.html',
        controller: 'AboutController'
    }).
    otherwise({
        redirectTo: '/home'
    });
});

//JSFiddle
```

The AngularJS example is very similar to the Backbone example except routes map to template URL's and controller classes (controllers are similar to Backbone's views in this usage).

Ember Routing Example

Below is a simple example of routing in Ember.

```
//html
<script type="text/x-handlebars" data-template-name="application">
    <h1> Ember Routing</h1>
    <div id="navigation">
        <a href="#">Home</a>
        <a href="#about">About</a>
    </div>
    <br />
    {{outlet}}
</script>
<script type="text/x-handlebars" data-template-name="home">
    <h2> Home Page </h2>
</script>
<script type="text/x-handlebars" data-template-name="about">
    <h2> About Page </h2>
</script>

//javascript
App = Ember.Application.create();

App.Router.map(function() {
    //first paramater refers to the template
    this.route('home', { path: '/' });
    this.route('about');
});

//JSFiddle
```

Again, very similar to the others except with Ember.js the first parameter to the router's "resource" object is a route name and the second is the URL. The order of these parameters confused me at first until someone pointed out that the path parameter is optional and frequently can be set by convention as it is with the "about" page in the example. Also, templates are required to make this simple routing example work but I'll go over them in a later section. For now it's enough to know the templates get placed in the {{outlet}}. It's worth noting Ember's router allows advanced features such as nesting of templates but here I'm just trying to show how similar in functionality it is to the other libraries.

Just the Routing

Most frameworks include a routing solution but Knockout.js focuses on data binding as discussed previously and recommends a best of breed solution with a JavaScript library whose only feature is routing. Common single-purpose libraries used for routing include SammyJS and HistoryJS. In summary, SammyJS works with browsers that support HTML5 History API as well as URL hashes for back level browsers (IE 8 and lower) . HistoryJS is smaller but only supports the HTML5 History API .

Data binding

Data binding allows changes in the model data to be updated in the view and/or changes in the view to be automatically updated in the model without additional code. One-way data binding generally indicates changes to the model are propagated to the view. Two-way data binding adds the ability for view changes to immediately be shown on the model. Data binding eliminates a lot of boilerplate code developers write and frees the developer to focus on the unique problems in the application.

AngularJS Data Binding Example

Below is a simple example of two-way data binding in AngularJS. Typing in the input field will show the entered text after the welcome message.

```
//html
<h1>Simple Data Binding with AngularJS</h1>
<br />
<div ng-app>
    Name: <input type="text" ng-model="name" />
    <br /><br />
    Welcome to AngularJS {{name}}
</div>

//javascript
//No JavaScript needed aside from the reference to the AngularJS script.

//JSFiddle
```

Knockout Data Binding

Knockout really just focuses on data binding and is used as part of a best of breed solution with other frameworks and libraries such as Durandal for screen management and composition and History.js or Sammy.js to handle the routing. Here is an example of data binding in Knockout.

```
//html
<div class="liveExample">

    <p>Enter name and tab out of field</p>
    <p>Name: <input data-bind='value: name' /></p>
    <p>Welcome to KnockoutJS <span data-bind='text: name'></span></p>

</div>

//javascript
// Here's my data model
var ViewModel = function(name) {
    this.name = ko.observable(name);
};

ko.applyBindings(new ViewModel("")); // This makes Knockout get to work

//JSFiddle
```

Backbone Data Binding

Backbone doesn't have automatic data binding but it is possible to do manually. In practice, I've found one-way data binding which updates the view when changes are made to the model to be extremely useful. Real-world use cases of data binding from the view to the model are less common (for example, you want a live preview of how text will look as the user types text in an editor). It can be helpful to have binding from the view to the model but frequently the view change is initiated from user input and you want to do other logic before or in addition to actually changing the model. For example, a specific use case would be validating input fields before changing or filtering a list. Below is a simple example where code has been implemented to bind both ways.

```
//html
<h1>Manual Data binding in Backbone.js</h1>
<p>Enter a name in the input below</p>
<div id="message-container">
    <input id="name" />
    <br />
    <span id="message"></span>
</div>

<script type="text/template" id="message-template">
    Welcome to Backbone  <%=name%>
</script>

//javascript
var MessageView = Backbone.View.extend({
    template: _.template($('#message-template').html()),
    events: {
        'keyup #name': 'updateModel'
    },
    updateModel: function(event) {
        this.model.set({name:$("#name").val()});
    },
    initialize: function() {
        this.listenTo(this.model, "change", this.render);
        this.render();
    },
    render: function() {
        this.$('#message').html(this.template(this.model.toJSON()));
        return this;
```

```
    }
});

var person = new Backbone.Model({name:''});
messageView = new MessageView({el:   $('#message-container') ,model:
person});

//JSFiddle
```

In summary, you listen for a change event on the model and call the view's render property to get the model to update the view. Similarly, you listen for "keyup" on an input and change the model by getting the value out of the input with jQuery and setting it on the model to have the view update the model. This example should give you a feel for how much code it takes to get data binding working. It is also worth noting that there are numerous plug-ins that add support for data binding to Backbone.

Ember Data Binding Example

Data binding in Ember looks like this:

```
//html
<script type="text/x-handlebars" data-template-name="application">
    <h1>EmberJS data binding</h1>
    {{outlet}}
</script>
<script type="text/x-handlebars" data-template-name="index">
    Name: {{input type="text" value=name }}
    <br />
    <br />
    Welcome to Ember {{name}}
</script>

//javascript
App = Ember.Application.create({});

App.IndexRoute = Ember.Route.extend({
    model: function(){
        return{
            name: ''
        };
    }
});

//JSFiddle
```

Ember uses the familiar handlebars for templating but the framework also includes "input helpers" to bind common form input fields. The curly braces {{ replace the angle brackets < on the input in this example and the "name" property has no quotes so the helper knows to bind it. Ember does support two-way binding with very little code similar to AngularJS.

Templates/Views

Templates can be entire pages of HTML but more commonly are smaller fragments of HTML with data binding placeholder expressions included for dynamic data. They can be logic-less with the philosophy that there should be little to no logic in your views or some allow you to embed JavaScript directly in the template. Templates can be DOM-based and use the DOM to dynamically insert dynamic data or string-based treating the HTML as strings and replacing the dynamic parts.

AngularJS Template Example

Here is a simple templates example in AngularJS.

```
//html
<script type="text/ng-template" id="embedded.home.html">
    <h1> Home </h1>
    {{greeting}}
</script>
<script type="text/ng-template" id="embedded.about.html">
    <h1> About </h1>
    {{content}}
</script>
<div>
    <div id="navigation">
        <a href="#/home">Home</a>
        <a href="#/about">About</a>
    </div>

    <div ng-view></div>
</div>

//javascript
var templatesExample = angular.module('FunnyAnt.Examples.Templates', []);
templatesExample.controller('HomeController', function ($scope) {
    $scope.greeting = "Welcome to the application.";
});
templatesExample.controller('AboutController', function ($scope) {
    $scope.content = "As a software developer, I've always loved to build
      things...";
});

templatesExample.config(function ($routeProvider) {
    $routeProvider.
    when('/home', {
        templateUrl: 'embedded.home.html',
        controller: 'HomeController'
    }).
    when('/about', {
        templateUrl: 'embedded.about.html',
        controller: 'AboutController'
    }).
    otherwise({
        redirectTo: '/home'
    });
});

//JSFiddle
```

You'll notice this is very similar to the earlier routing example with some data binding added to show how templates can help in your application. Templates are all included in script tags in the main HTML file to make the example easy to follow and work in jsfiddle.net but templates can be external to the view in AngularJS by simply giving a valid file path to the templateUrl property when configuring the $routeProvider.

It's worth mentioning that the preferred way of handling templates in larger scale applications where performance is a concern is to concatenate and register your AngularJS templates in the Angular $templateCache at compile time with a build task such as this one .

Ember Template Example

Below is an example of templates in Ember.

```
//html
<script type="text/x-handlebars" data-template-name="application">
    <h1> Ember Templates</h1>
    <div id="navigation">
        <a href="#">Home</a>
        <a href="#about">About</a>
    </div>
    <br />
    {{outlet}}
</script>
<script type="text/x-handlebars" data-template-name="home">
    <h2> Home Page </h2>
    {{greeting}}
</script>
<script type="text/x-handlebars" data-template-name="about">
    <h2> About Page </h2>
    {{pagecontent}}
</script>

//javascript
App = Ember.Application.create({});

App.Router.map(function() {
    //first paramater refers to the template
    this.resource('home', { path: '/' });
    this.resource('about', {path: '/about'});
});

App.HomeRoute = Ember.Route.extend({
    model:function(){
        return{
            greeting: 'Welcome to the application.'
        }
    }
});

App.AboutRoute = Ember.Route.extend({
    model: function(){
        return{
            pagecontent: 'As a software developer, I have always loved to
            build things...'
        };
    }
});
//JSFiddle
```

An Ember route is an object that tells the template which model it should display. But it is not a URL as one might expect. I think of it as the most basic controller for your template and resource (URL) whose main job is to load the model. If you need to get fancy and store application state then you need a controller.

Ember and AngularJS do use different implementations of templating: string based and DOM based templates respectively but they are trying to solve the same problems. Furthermore, the interfaces and contracts they surface to the developer are very similar. More specifically, they both use handlebars syntax to do bindings and script tags or separate files to store the templates.

Backbone Template Example

Now, let's look at a simple example of templates in Backbone.

```
//html
<div id="navigation">
    <a href="#/home">Home</a>
    <a href="#/about">About</a>
</div>
      <div id="content">
</div>

<script type="text/template" id="home-template">
    <h1>Home Page</h1>
    <%= greeting %>
</script>

<script type="text/template" id="about-template">
    <h1>About Page</h1>
    <%= content %>
</script>

//javascript
var HomeView = Backbone.View.extend({
    template: _.template($("#home-template").html()),
    initialize: function () {
        this.render();
    },
    render: function () {
        this.$el.html(this.template({greeting:"Welcome to Backbone!"}));
    }
});

var AboutView = Backbone.View.extend({
    template: _.template($("#about-template").html()),
    initialize: function () {
        this.render();
    },
    render: function () {
        this.$el.html(this.template({content:"As a software developer,
        I've always loved to build things..."}));
    }
});

var AppRouter = Backbone.Router.extend({
    routes: {
        '': 'homeRoute',
```

```
        'home': 'homeRoute',
        'about': 'aboutRoute',
    },
    homeRoute: function () {
        var homeView = new HomeView();
        $("#content").html(homeView.el);
    },
    aboutRoute: function () {
        var aboutView = new AboutView();
        $("#content").html(aboutView.el);
    }
});

var appRouter = new AppRouter();
Backbone.history.start();

//JSFiddle
```

This is a modification of the routing example. But instead of the HTML being hard-coded in the view object's template property the markup is now in the HTML page inside a script tag with an id attribute (browsers ignore script tags with types they do not recognize such as text/template so the template won't be shown). To get the template (HTML fragment) we use a jQuery selector to find the element by the script tag's id and grab the inner HTML and then assign the HTML to the template property of the view object (it's just a string).

Given this approach of treating templates as strings of HTML, it's easy to imagine how a different template library can be substituted in a Backbone application by simply implementing the template property slightly differently.

For example to use the Handlebars templating library instead of Underscore's templating library we would update the view's template property as follows:

template: `Handlebars.compile( $("#home-template").html() ),`
...and then update the binding syntax used in the templates for example
{{greeting}}

With these few minor changes we have changed the template library used in the application.

```
//html
<div id="navigation">
    <a href="#/home">Home</a>
    <a href="#/about">About</a>
</div>
<div id="content">
</div>

<script type="text/template" id="home-template">
    <h1>Home Page</h1>
    {{ greeting }}
</script>

<script type="text/template" id="about-template">
    <h1>About Page</h1>
    {{ content }}
</script>

//javascript
var HomeView = Backbone.View.extend({
    template: Handlebars.compile( $("#home-template").html() ),
    initialize: function () {
        this.render();
    },
    render: function () {
        this.$el.html(this.template({greeting:"Welcome to Backbone!"}));
    }
});


var AboutView = Backbone.View.extend({
    template: Handlebars.compile( $("#about-template").html() ),
    initialize: function () {
        this.render();
    },
    render: function () {
              this.$el.html(this.template({content:"As a software developer,
               I've always loved to build things..."}));
    }
});

var AppRouter = Backbone.Router.extend({
    routes: {
        '': 'homeRoute',
        'home': 'homeRoute',
        'about': 'aboutRoute',
    },
    homeRoute: function () {
```

```
        var homeView = new HomeView();
        $("#content").html(homeView.el);
    },
    aboutRoute: function () {
        var aboutView = new AboutView();
        $("#content").html(aboutView.el);
    }
});

var appRouter = new AppRouter();
Backbone.history.start();

//JSFiddle
```

It's hard to draw the line between where some of these concepts end and the next ones start. When many people talk about templates they get into a given templating library's performance which has a lot to do with data binding previously discussed but also includes how the templates are loaded (Are they precompiled into JavaScript?). To add to the confusion, templates are basically HTML as mentioned before and many server-side MVC developers associate HTML with the term views or partials. However, views in Backbone and Ember are code, a JavaScript class that manages an HTML fragment, and generally don't have much markup in them except a reference to a template which is generally external to the view object.

Models

Models are the client-side version of what is commonly referred to as business objects, domain objects, or entities. As described by Jeremy Ashkenas, Backbone author, "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." Ashkenas has been careful to say these are not going to be a one-to-one mapping with your Active Record models on the server but a smaller collection of them on the client with additional properties that are useful to the user interface such as a count. In general, the idea behind models in client-side MV* frameworks is to establish a central point for the data in the application as well as any behavior that should be encapsulated with that data. Sserver-side MVC plus jQuery architectures commonly store the model data in the DOM. By having model objects the goal is to remove that data and state from the DOM and put it in a common place where it can be reused.

Backbone Model Example

Our earlier example of data binding showed off models. In summary, models hold data and keep it out of the DOM and emit events such as "change" which allows numerous views to react accordingly and update the user interface everywhere it is needed. So now you have one source of the truth and it's not the user interface.

```
//html
<h1>Models in Backbone.js</h1>
<p>Enter a name in the input below</p>
<div id="message-container">
    <input id="name" />
    <button id="button">Save</button>
    <br /> <span id="message"></span>

    <br />
</div>
<div id="name-container"></div>
<script type="text/template" id="message-template">
    Welcome to Backbone <%= name %>
</script>
<script type="text/template" id="name-template">
    Person: <%= name %>
</script>

//javascript
var MessageView = Backbone.View.extend({
    template: _.template($('#message-template').html()),
    events: {
        'click #button': 'updateModel'
    },
    updateModel: function (event) {
        this.model.set({
            name: $("#name").val()
        });
        $("#name").html('');
    },
    initialize: function () {
        _.bindAll(this, 'render');
        this.listenTo(this.model, "change", this.render);
    },
    render: function () {
        this.$('#message').html(this.template(this.model.toJSON()));
        return this;
    }
```

```
});

var NameView = Backbone.View.extend({
    template: _.template($('#name-template').html()),
    initialize: function () {
        _.bindAll(this, 'render');
        this.listenTo(this.model, "change", this.render);
    },
    render: function () {
        this.$el.html(this.template(this.model.toJSON()));
        return this;
    }
});

var Person = Backbone.Model.extend({
    defaults: {
        name: ''
    }
});

var person = new Person();

var messageView = new MessageView({
    el: $('#message-container'),
    model: person
});

var nameView = new NameView({
    el: $('#name-container'),
    model: person
});

//JSFiddle
```

I've modified the data binding example from earlier by adding a new template and view that looks at the same Person model object. Previously, I declared the Person model on the fly to keep things simple but now I've added the call to the Backbone.Model.extend() method to demonstrate how you create a prototype for a model that can be used over and over similar to classes in classical languages. Notice how both views are listening to the same person model object (the change event) and updating themselves. By having this single source of data the numerous calls to specific DOM elements can be encapsulated in their own tidy views and one model can serve them all.

Notice how the get and set accessors need to be used for the change event to be fired in the example above...it is not uncommon to forget this convention.

AngularJS Model Example

The idea of one model that is the truth about the "state'" in your application exists in AngularJS but Angular allows you to use plain old JavaScript objects as your model and then adds watchers "under the hood" to any property that is added to the $scope object and declaratively data bound in the view with the directive "ng-model." These watchers then automatically alert other parts of the application that are bound to that same model and these DOM elements know how to update themselves. So there is a lot more "magic" going on but the AngularJS team argues it's "good magic".

Here is the updated AngularJS data binding example showing two parts of the view being updated.

```
//html
<h1>Simple Data Binding with AngularJS</h1>
<br />
<div ng-app>
    Name: <input type="text" ng-model="person.name" />
    <br /><br />
    Welcome to AngularJS {{person.name}}
    <br />
    Person: {{person.name}}
</div>

//javascript
//No JavaScript needed for this example
```

AngularJS creates the person object on the scope for you when you data bind with the ng-model attribute so the example is very terse and includes no JavaScript.

This may not be the fairest comparison because we haven't created multiple controllers to manage the different parts of the view as we did with Backbone but regardless it will end up

being significantly less code with AngularJS.

Data Storage

AngularJS Data Example

AngularJS handles data in two different ways. First, by providing support for manual AJAX calls in a very similar way to jQuery $.ajax functionality via $http. In addition, if your backend is a strictly RESTful service, AngularJS provides a $resource class that makes calls to the RESTful service extremely terse.

$http example

```
//javascript
app.factory('myService', function ($http) {
    return {
        getFooOldSchool: function (callback) {
            $http.get('foo.json').success(callback);
        }
    }
});

app.controller('MainCtrl', function ($scope, myService) {
    myService.getFooOldSchool(function (data) {
        $scope.foo = data;
    });
});
```

$resource example

```
//javascript
var Todo = $resource('/api/1/todo/:id');

//create a todo
var todo1 = new Todo();
todo1.foo = 'bar';
todo1.something = 123;
todo1.$save();

//get and update a todo
var todo2 = Todo.get({ id: 123 });
todo2.foo += '!';
todo2.$save();
```

```
//delete a todo
Todo.$delete({ id: 123 });
```

Backbone Data Example

Backbone assumes you are interacting with a RESTful API but allows you to override one method `Backbone.sync()` if not. You tell your model where the resource is on the server (the URL) and then you can just call save.

```
//javascript

var UserModel = Backbone.Model.extend({
    urlRoot: '/user',
    defaults: {
        name: '',
        email: ''
    }
});
var user = new Usermodel();
// Notice that we haven't set an `id`

var userDetails = {
    name: 'Craig',
    email: 'craigmc@funnyant.com'
};
// Because we have not set an `id` the server will call
// POST /user with a payload of {name:'Craig', email: 'craigmc@funnyant.com'}
// The server should save the data and return a response containing the new
`id`
user.save(userDetails, {
    success: function (user) {
        alert(user.toJSON());
    }
});
```

Ember Data Example

Ember has Ember Data which is not technically part of the core framework but is shooting to provide a more robust data persistence/data storage story. It provides many of the facilities you'd find in server-side ORMs like ActiveRecord, but is designed specifically for the unique environment of JavaScript in the browser. At the time of writing however the Ember Core Team is close to releasing v1.0 but has not and most Ember projects simply use the $.ajax methods in jQuery just as AngularJS uses $http in the above examples.

We're all in the Same Gang

This chapter broke down JavaScript MV* frameworks into features to provide insight into what functionality is provided by these frameworks and to bring readers to the realization that they are actually very similar. Once you understand the features and how they fit together it becomes much easier to quickly learn multiple frameworks and find the right one for your project.

CHAPTER 5

EXAMPLE APPLICATION

Application Overview

To demonstrate the various JavaScript MV* frameworks and libraries you will build a clone of some of the functionality in the popular project management software Basecamp. To begin you'll go through a step by step tutorial of creating a project list with add, edit and delete features.

After you get comfortable with the framework concepts we'll discuss other features including support for multiple lists for each project and multiple "todo" tasks on each list.

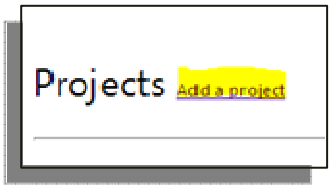
The objects needed and their hierarchy is as follows:

- Project (name, description)
  - List (name, description, projectid)
    - Todo (description, completed, duedate)
      - AssignedUser

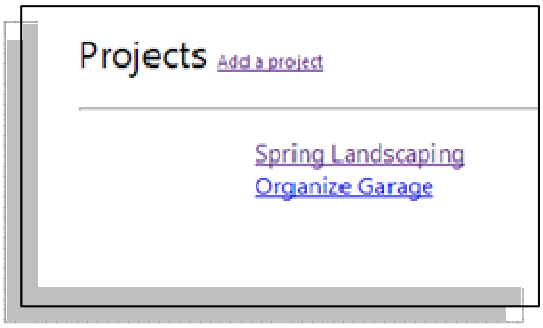
At the top of the hierarchy is a list of projects. Each project can have multiple lists. Each list can have multiple todo items. Each todo item can be assigned to a user, assigned a due date, and/or marked as completed.

Below are some screenshots to demonstrate each of these actions.

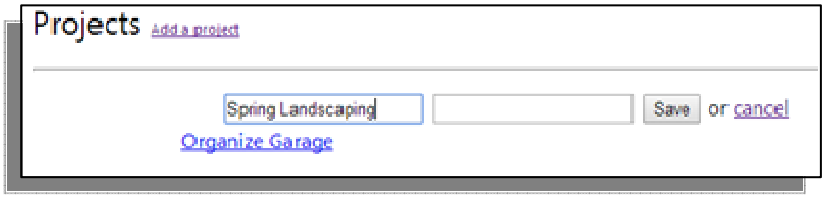
To add a project:



To view a list of projects:



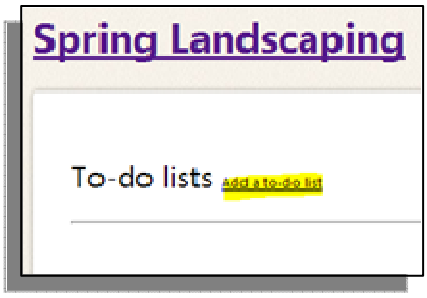
To edit a project you click the edit link on the project list next to the project's name.

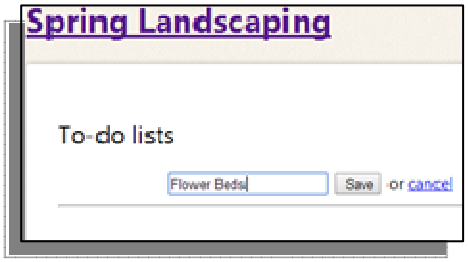


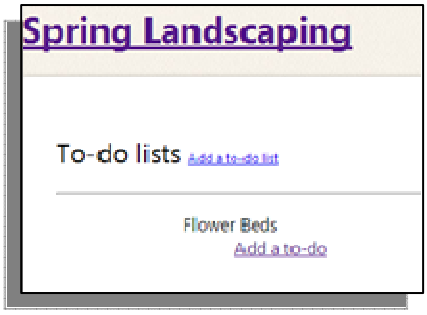
And to remove a project you click the "delete" link on the project list next to the project's name (see above).

To see the lists and todos for a project you click the project's name on the project list (see above).

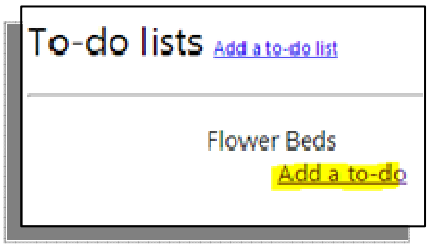
To add a list you click "Add list" from the project details page.

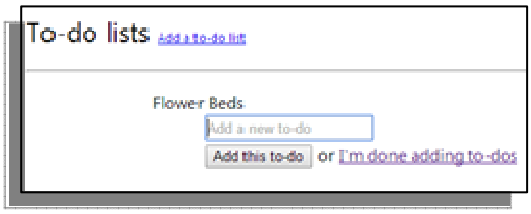


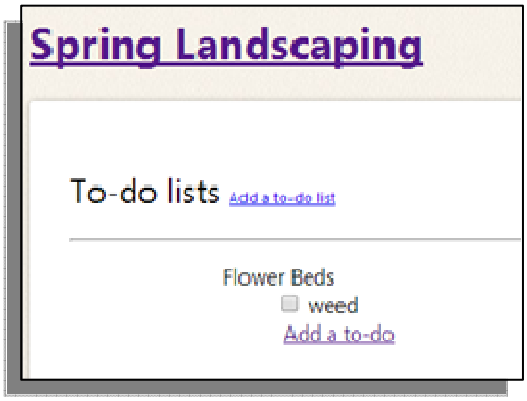




To add todo items from the project details page you click "Add a to-do list"

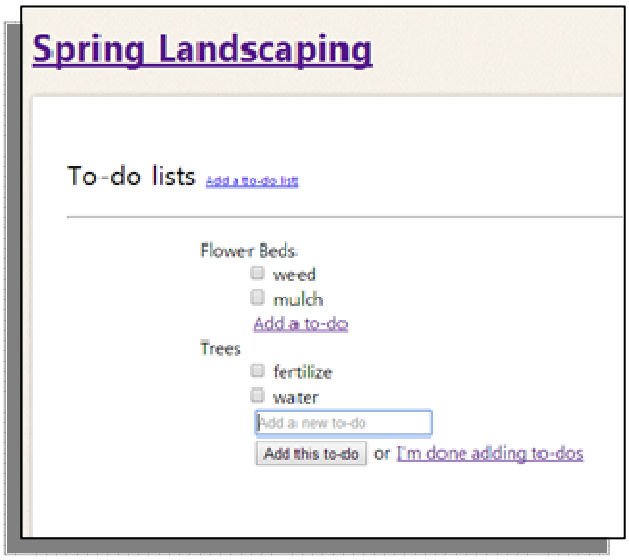




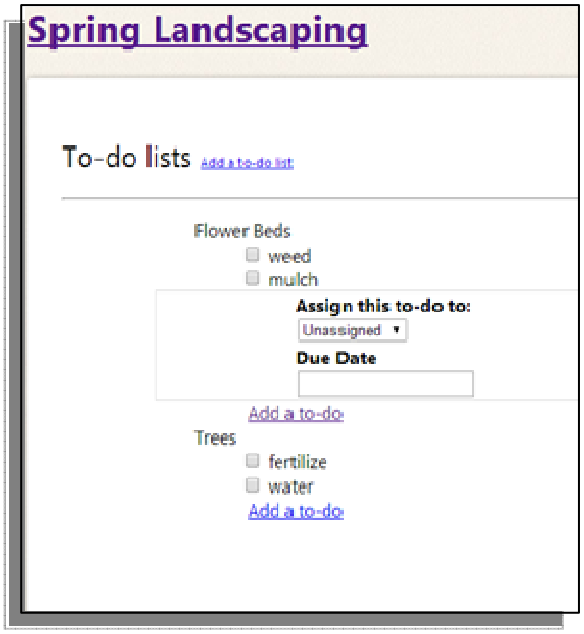


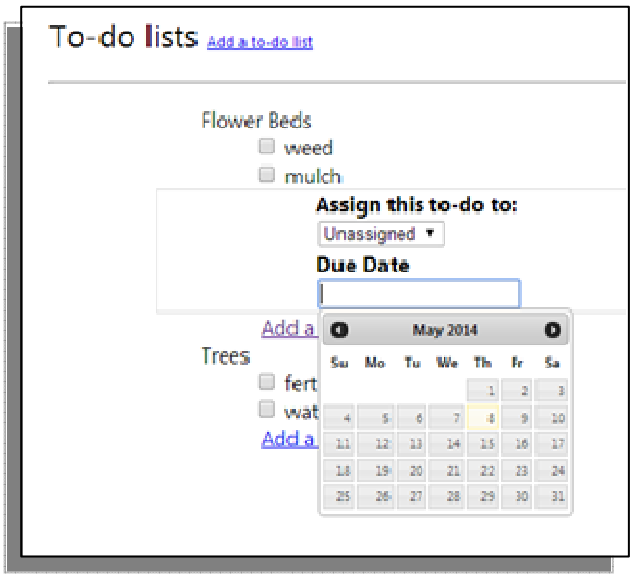
Editing or deleting lists and todo items can be done by clicking "edit" or "delete" next to the descriptions in the list on the project details page in the same way projects work.

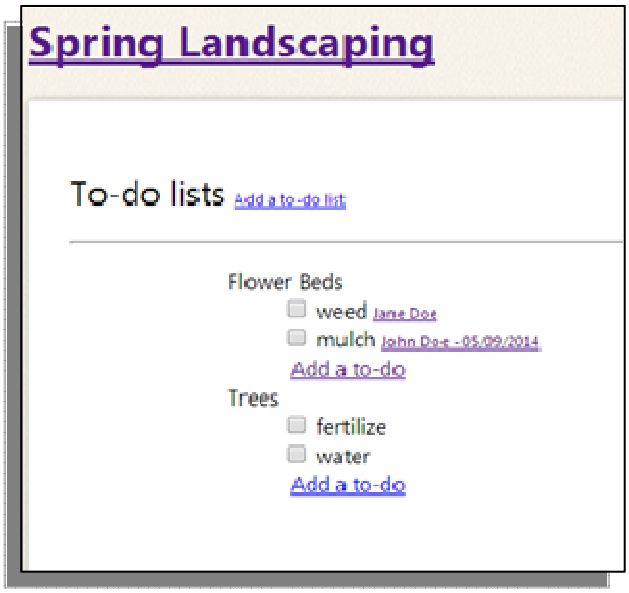
Multiple lists and todo items can be added.



In the AngularJS example application you can also assign todos to a user and give them a due date.






Spring Landscaping
To-do lists
Add a to-do list
Flower Beds
weed
Jane Doe
mulch
John Doe - 05/09/2014
Add a to-do
Trees
fertilize
water
Add a to-do

How to Debug

One important question that comes up is how to debug JavaScript applications. One major evolution in your comfort in using JavaScript for client-side development will be getting out of your familiar Integrated Development Environment (IDE) (Visual Studio, NetBeans, Eclipse, etc...) and opening up a web browser to debug the source code. It makes sense to debug in your browser because that is where the code is running. I suggest using Google Chrome and you'll soon leave behind alerts for a better world.

Debugging in Chrome

Open a Chrome Web browser. If you don't have it installed go here and do it because we will be using it for debugging JavaScript throughout this book. It takes about 30 seconds to install and can be installed at most corporations even if you don't have administrative access to install applications.

Go to the list icon on the far right of your browser address bar and choose Tools→ Developer Tools. After the developer tools window opens docked at the bottom of your screen, choose the console tab on the far right.

OR

Use the shortcut key Ctrl+Shift+J on a Windows PC or Cmd+Option+J on a Mac

Note that the shortcut key Ctrl+Shift+I on a Windows PC or Cmd+Option+I on a Mac opens the Chrome Developer tools but with the "Elements" tab as the default.

If you need to set a break point simply navigate to the "Sources" tab in Chrome Developer

Tools and open the tree of JavaScript files on the left and find the one you want to debug. Open the file and click on the line number where you want to set the breakpoint and you are set. Refresh the page or trigger your code by interacting with the user interface and your breakpoint will be hit.

Setting up the Examples

The example applications are all in one GitHub repository projectmanagemvc. The folder structure is as follows:

\projectmanagemvc\angularjs\projectmanage\
\projectmanagemvc\backbone\projectmanage\
\projectmanagemvc\ember\projectmanage\

You'll need to setup a web site for each framework using your preferred local web server and pointing it at the "projectmanage" directory under that framework's folder (for example \projectmanagemvc\angularjs\projectmanage\) and then requesting the root (http:\\localhost:81\) or include index.html if that is not set as a default page on your web server.

Summary

You should now have a feel for what the example application will look like and what functionality if will include. And you now know where to clone the Git repository and get the example projects set up on your computer. Lastly, you've taken the time to get familiar with Chrome's JavaScript debugging and are hopefully getting more comfortable with it if you weren't already using it.

CHAPTER 6

Backbone Example Application

Overview

Let's review all the pieces we'll need to build the project related functionality:

- A Project model to describe each individual project
- A Project model to describe each individual project
- A Projects collection to store and persist projects
- A way to create a project
- A way to display a project
- A way to display a listing of projects
- A way to edit a project
- A way to delete a project
- A way to update the URL as the application state changes to add a project

Let's get started.

Static HTML

All of our HTML will be in a single file index.html.

Placing all the HTML in one file was done to make the example easy to follow but as mentioned previously, in a production application developers usually put the templates in separate files and pre-compile them all into minified JavaScript. The pre-compilers usually look at all files in a directory or all files with a given file extension for example .hbs for Handlebars templates.

Header and Scripts

One of the nicest things about JavaScript frameworks is that including them in your project is as simple as downloading a few JavaScript files and including some script tags in your main index.html.

Backbone depends on jQuery (or another DOM manipulation library such as Zepto.js) and Underscore.js. In addition, we'll include the Backbone localStorage plugin which we'll discuss in more detail when we set-up the project collection object.

```
// /index.html
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/site.css" rel="stylesheet">
</head>
<body>
    <script src="/js/libs/jquery/jquery-1.10.2.min.js"></script>
    <script src="/js/libs/underscore/underscore-min.js"></script>
    <script src="/js/libs/backbone/backbone.js"></script>
    <script type="text/javascript"
     src="/js/libs/backbone/backbone.localStorage.js"></script>
</body>
</html>
```

The scripts are at the end of the page before the body tag so they don't block the page from loading. I also included the Bootstrap framework to make the style sheet easier to code in the example.

Templates

The `index.html` page is essentially the "shell" for the application and other templates are loaded into it. As described previously you use familiar jQuery selectors (referencing the template script tag by its id) to get the HTML out of the script tags and then set it into the content div on the page using `$("#content").html(templateHTML)`.

Below is the markup with the "shell" content div tag **bolded** and the templates *italicized* so you can visualize how it works. We'll fill in the templates in the next step.

```
<!DOCTYPE HTML>
<html>
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/site.css" rel="stylesheet">
</head>
<body>
    <div id="content" class="container">
    </div>
    <script type="text/template" id="projects-template">
       <!--markup goes here-->
    </script>
    <script type="text/template" id="project-add-template">
       <!--markup goes here-->
    </script>
    <script type="text/template" id="project-template">
       <!--markup goes here-->
    </script>
    <script src="/js/libs/jquery/jquery-1.10.2.min.js"></script>
    <script src="/js/libs/underscore/underscore-min.js"></script>
    <script src="/js/libs/backbone/backbone.js"></script>
    <script type="text/javascript"
     src="/js/libs/backbone/backbone.loc alStorage.js"></script>
</body>
</html>
```

Templates with HTML

Here are what the project related templates look like with HTML.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>Index</title>
    <link href="/css/bootstrap.min.css" rel="stylesheet">
    <link href="/css/site.css" rel="stylesheet">
</head>
<body>

    <div id="content" class="container"></div>

    <script type="text/template" id="projects-template">

        <h1><a href="/">Projects</a></h1>
        <div id="add-project" class="row">
        </div>

        <br class="clearfix" />
        <hr />
        <div class="row">

            <ul id="projects" class="list-unstyled"></ul>

        </div>

    </script>

    <script type="text/template" id="project-add-template">

        <div class="col-md-6 view-project">
            <a class="btn btn-primary add pull-right"
            href="#project/add">Add Project</a>
        </div>

        <div class="col-md-6">
            <form class="add-project form-vertical well">
                <div class="form-group">
                    <label>Project Name</label>
                    <input class="name form-control" type="text"
                     placeholder="enter project name"
```

```
                    value='<%=name%>' />
                </div>

                <div class="form-group">
                    <label>Project Description</label>
                    <textarea class="description form-control"
                     rows="6"
                     placeholder="enter description here">
                     <%=description%>
                     </textarea>
                </div>
                <button class="project-save btn btn-primary">
                 Save
                </button>
                <a class="cancel" href="#projects">cancel</a>
            </form>
        </div>

    </script>


    <script type="text/template" id="project-template">

        <div class="col-md-6 view-project">
            <a href="#project/<%=id%>/lists"><%=name%></a>
            <a class="edit" href="#project/edit/<%=id%>">edit</a>
            <a class="remove" href="#project/remove">remove</a>
        </div>

        <div class="col-md-6">
            <form class="edit-project form-vertical well">
                <div class="form-group">
                    <label>Project Name</label>
                    <input class="name form-control"
                     type="text" placeholder="enter project name"
                     value='<%=name%>' />
                </div>

                <div class="form-group">
                    <label>Project Description</label>
                    <textarea class="description form-control"
                     rows="6" placeholder="enter description">
                     <%=description%>
                     </textarea>
                </div>
```

```
                <div class="form-group pull-right">
                    <button class="project-save btn btn-primary">
                    Save
                    </button>
                    <a class="cancel" href="#project/view">cancel</a>
                </div>
                <br class="clearfix" />
            </form>
        </div>

    </script>

    <script src="/js/libs/jquery/jquery-1.10.2.min.js"></script>
    <script src="/js/libs/underscore/underscore-min.js"></script>
    <script src="/js/libs/backbone/backbone.js"></script>
    <script type="text/javascript"
       src="/js/libs/backbone/backbone.localStorage.js"></script>
    <script src="js/models/project.js"></script>
    <script src="js/collections/projects.js"></script>
    <script src="js/views/project-view.js"></script>
    <script src="js/views/project-add-view.js"></script>
    <script src="js/views/projects-view.js"></script>
    <script src="js/app.js"></script>
    <script src="js/routers/router.js"></script>
</body>
</html>
```

Notice the Underscore micro-template expressions <%= %> in the templates which will be replaced in the view with model data.

Project Model

We'll need a model object to represent a project. You can imagine a much more elaborate project object but we'll keep things simple and focused on Backbone.

```
// js/models/project.js

var app = app || {};

app.Project = Backbone.Model.extend({
    defaults: {
        name: '',
        description: '',
        lists: []
    },
    initialize: function () {
        this.on("invalid", function (model, error) {
            console.log(error);
        });
    },
    validate: function (attrs) {
        if (!attrs.name) {
            return "Project Name is required.";
        }
    },
});
```

Note: The lists array is there to hold multiple child lists of "todo" items and will not be used in this initial section.

This model object serves the same purpose as an abstract base class does in object-oriented languages but is not a class. It is a constructor function with a prototype property that holds all the boilerplate stuff you want included every time you create an object including the default properties you just defined.

Load the page and type this in the console to see the defaults on the prototype:
app.Project.prototype.defaults

To be specific when you call Backbone.Model.extend() the object you get back is a function, more specifically a constructor function. When you use a constructor function in JavaScript with the new keyword it creates and returns a new object that has a reference to the constructor that created it via the object.constructor property and the constructor has a prototype Backbone uses to store all the custom stuff you added when you extended the object from Backbone.Model.

Why didn't you tell me?

JavaScript is case-sensitive and objects in Backbone follow the convention of being upper-case and methods use lower casing. Unfortunately I commonly type Backbone.Model.Extend (leaving extend upper cased) and receive a "has no method 'Extend'" error. So remember methods are lower-cased.

Change Tracking

This object also has a lot of other useful stuff you would want on a business object in your system such as change tracking that can fire an event for any property that changes. You can listen for changes with the .on method...the "change:name" string follows the pattern "event:property".

```
var project = new app.Project();

project.on("change:description", function (list) {
    console.log("Changed descripton to: " + list.get("description"));
});

project.set("description", " Project Runway");

//Outputs Changed name to: Project Runway
```

Notice that properties on the model need to be accessed using the .get method and set using the .set method. Using these get and set methods allows backbone to implement the observer pattern and watch for changes to the properties and notify other parts of the application. This is a strength of the backbone.js library in that it allows efficient tracking of

model properties but is a weakness in that when using the framework you are required to use the special .get and .set methods which can make your code more difficult to read and maintain.

Validation

Another feature Backbone.Model gives you is validation via a validate method that returns nothing if the model object is valid but returns an error message or messages if the object is invalid.

Let's implement a validation rule that a project name is required. First we add a validate function to the Model as well as listen for the "invalid" event in the initialize function of the model.

```
var project = new app.Project({ name: 'Craig' });

//set description back to empty string so validation will fail
project.set('name', '', { validate: true });

//logs to console "Project name is required." Because of code in initialize
in app.Project
```

We'll look at how to check if a model is valid and display this validation message in the user interface when we implement the views.

The initialize function passed to `Backbone.Model.extend()` can be thought of as a place to put code you would normally put in the constructor of a class in an object-oriented language.

What is .extend() doing?

As a quick sidebar run the following code in your browser console.

```
app.Project.validate = function (attrs) {
    return "invalid";
};
```

What is logged to the console? Still "Project Name is required" because we changed the app.Project function object but not the app.Project prototype and the prototype is what is used as the template for new object instances. Now run this code in your browser's console:

```
app.Project.prototype.validate = function (attrs) {
    return "invalid";
};
```

The phrase "invalid" is logged to the console this time. Please refresh the page to remove this extra validate method before continuing.

The important point here is everything you declare in `Backbone.Model.extend()` or `Backbone.View.extend()` gets put on the prototype of the constructor function returned from those methods.

Persistence

Models are also the key to persistence in the application. If you call save on your Backbone model object, it then calls `backbone.sync` and `"syncs"` the model to the service.

There is a lot of other goodness in the Backbone.Model but these are the most used features and the main ones we will use while building our application.

Project Collection

Backbone collections are an in-memory collection of models that you use in your application's interface. Collections also have methods that allow you to fetch these models from your persistence which by default is assumed to be a RESTful server-side API.

The collection in this example uses the localStorage adapter to override Backbone's default sync implementation and store them in HTML5 localStorage. This allows our data to be persisted between page requests and after closing the web browser.

```
// js/collections/projects.js

var app = app || {};

app.Projects = Backbone.Collection.extend({
    model: app.Project,
    localStorage: new Backbone.LocalStorage("PersistedProjects"),
});


// collections-example.js

var project = new app.Project({ name: 'original project' });
console.log(project.toJSON());

var projects = new app.Projects();

//listen for project to be added
projects.on('add', function (project) {
    console.log('Added: ' + JSON.stringify(project.toJSON()));
});

projects.add(project);
//Logs: Added: {"name":"original project","description":"","lists":[]}

//listen for project name to change
projects.on('change:name', function (project) {
    console.log('Changed name to: ' + JSON.stringify(project.get("name")));
});
project.set("name", "new project");
//Logs: Changed name to: "new project"

//listen for project to be removed
```

```
projects.on('remove', function (project) {
    console.log('Removed: ' + JSON.stringify(project.toJSON()));
});

projects.remove(project);
//Logs: Removed: {"name":"new project","description":"","lists":[]}

var project1 = new app.Project({ name: 'project 1' });
var project2 = new app.Project({ name: 'project 2' });
var project3 = new app.Project({ name: 'project 3' });
projects.add([project1, project2, project3]);
console.log('Collection now has: ' + projects.length + " items");
//Logs: Collection now has: 3 items

//save two of them to persistence/localStorage
project1.save();
project2.save();

//reload from persistence/localStorage
projects.fetch();
console.log('Collection now has: ' + projects.length + " items");
//Logs: Collection now has: 2 items

//removes all models from persistence/localStorage
project1.destroy();
project2.destroy();
```

In addition to showing you how to create a collection, we put together some examples using the projects collection to better demonstrate how it works with the model. You can see the running example code by opening collections-example.html at the root of the Backbone project.

One common use for a collection is to listen to events such as "add", "change", "remove", and "fetch" inside a Backbone.View object and then take appropriate action which is frequently to update or render the view again. In our example, we log to the browser console when we receive an event but you will see in the next section how our views use the events to repaint themselves.

Another use for collections is to assist with persistence and retrieval of models. More specifically, the URL property is generally set on the collection and/or model and when the

save or destroy methods are called on the model a call to a server-side API is made. In our examples, the localStorage property is set on the collection (instead of the more common URL property) and when save or destroy is called models are saved or deleted from HTML5 localStorage. This works because the localStorage plugin overrides the Backbone.sync method (which is the common method for all persistence operations: adds, changes, deletes, etc...) and instead of making AJAX requests it makes calls to the localStorage API of the browser to persist or retrieve the data.

Later we'll address syncing to the server in more detail but this demonstrates the power of thin server architectures where the entire front-end for an application can be prototyped, iterated, and solidified to the end user's delight before or while server-side work is being done on a web service API (to work out of the box with Backbone it should be RESTful) to persist the model data to a database.

Backbone Views

A Backbone.View is responsible for managing a single root DOM element and all its child elements. In the projects section of the example application we have three views:

1. A projects-view.js that manages the <ul> that contains the list of projects as well as several child views including one that manages adding a new project and several instances of a view that manages each project item in the list. These child views are described in more detail below.
2. Several project-view.js instances that manage each project's individual <li> within the <ul>. These individual item views also manage an edit form for changes to each project and a "remove" link to delete the item.
3. A project-add-view.js that manages the <form> element for adding a project as well as an "add project" button which toggles whether the form is visible.

A lot of people get hung up at this point trying to compare Backbone views to a familiar concept from their server-side web MVC framework of choice Rails, Cake, JSF, ASP.NET MVC etc. In general, these discussions can be more academic than helpful in learning Backbone. To summarize, Backbone views are objects or code and not HTML as views are in all web server-side MVC frameworks. If you want something to associate it with in a server-side web framework (which is where most people have experience with MVC frameworks) then I would say they are closest to ASP.NET Web Forms User Controls or Java Facelets.

Backbone views really only have three properties and two functions that you need to learn in order to understand them. Furthermore, the two methods tend to follow some common steps in their implementation which I will outline for you. Some people consider these steps boiler-plate code others a flexible architecture.

Let's look at our first view and talk through these few properties and functions and the

common steps taken with each.

Project View

The first view we'll examine is the one responsible for each project's line item <li> in the project list <ul>.

```
// js/views/project-view.js

app.ProjectView = Backbone.View.extend({
    template: _.template($("#project-template").html()),
    tagName: "li",
    className: "row",
    events: {
        "click .cancel": "cancel",
        "click  .project-save": "save",
        "click .remove": "clear"
    },
    initialize: function () {
        _.bindAll(this, "render", "editing", "cancel", "save", "clear");
        this.listenTo(this.model, "change", this.render);
        this.listenTo(this.model, "destroy", this.remove);
        this.listenTo(app.projects, "editing:project", this.editing);
    },
    render: function () {
        this.$el.html(this.template(this.model.toJSON()));
        this.$name = this.$(".name");
        this.$description = this.$(".description");
        return this;
    },
    editing: function (args) {
        if (args.id != this.model.id) {
            return;
        }
        this.$el.addClass("editing-project");
        this.$name.focus();
    },
    cancel: function (e) {
        e.preventDefault();
        this.$el.removeClass("editing-project");
        app.appRouter.navigate('');
    },
    save: function (e) {
        e.preventDefault();
        this.model.set({ name: this.$name.val(),
                         description:  this.$description.val() });
        if (!this.model.isValid()) {
            alert(this.model.validationError);
```

```
            return;
        }
        this.model.save();
        this.$el.removeClass("editing-project");
        app.appRouter.navigate('');
    },
    clear: function () {
        this.model.destroy();
    }
});
```

The "el" View Property

The "el" property in a view is simply a reference to the root DOM element that is being managed by that particular view. The common pattern is to set the inner HTML of the element with a template's content interspersed with model data in the render function.

Working with el?

You've probably seen backbone.js examples that define and work with the property el several different ways. Here are some examples to clarify what you can and can't do.

`el: "#content"` //works, preferred b/c terse

`el: $("#content")` //works

`$el: $("#content")` // doesn't work, $el is just intended as a shorthand property getter for el wrapped as a jQuery wrapped element collection not a setter

`$(this.el).html("test");` //works

`this.$el.html("test");` //works, preferred b/c terse

//renders

```
<div id="content">
    test
</div>
```

Why `.append()` sometimes not `.html()`

The most common use case for .append is when looping through a collection and adding a sub item view for each element in the collection as shown below.

```
this.$("#projects").append("<li>another project</li>");
```

//renders

```
<div id="content">
    <ul id="projects">
        <li>another project</li>
    </ul>
</div>
```

The "template" View Property

The template property is where a compiled copy of the view's template can be stored so it can be reused multiple times if needed within the view. The templating engines "compile" the HTML from the template into JavaScript.

```
app.ProjectView = Backbone.View.extend({
    template: _.template($("#project-template").html()),
    //code omitted here for clarity/brevity
});
```

The "events" View Property

The "events" view property is an object literal with key value pairs of event plus jQuery selectors on the left as the key and view functions as the values. The true genius in this approach is the selectors attach their events to the root element in the view scoping them to a frequently much smaller portion of the page and making them significantly more efficient in those cases.

```
events:{
    "click .cancel":"cancel",
    "click  .project-save":"save",
    "click .remove":"clear"
```

```
}
```

The "initialize" View Function

The initialize method is the constructor of the view object. As mentioned earlier, there is a set of steps that are commonly followed when implementing a constructor. Steps are omitted depending on whether the view represents an item (project-view.js) versus a list of items or `collection (projects-view.js)`. The steps are:

1. Use the Underscore utility libraries function `_.bindAll` to set "this" to be the view in all other functions in the view where "this" is used or might be used.

```
_.bindAll(this,"render","editing","cancel","save","clear");
```

The first argument to `_.bindAll` is the object you want to be the context. Almost always you'll pass "this" to `_bindAll` so that the context will be the view itself. The remaining arguments are a comma separated list of all the functions where you want the context to be the view object.

This step is important because it's common to have a click event wired up to a function in the view via the "events" property of the view and when the user clicks "this" in the event handler (the function on the view) will be the element that was clicked. This is a problem because you've lost access to all other functions and properties on the view that you might want to use to handle the event including this.model, this.collection. Furthermore, it's easy to get a reference to that button either by setting a property reference to it in initialize or render which keeps you from repeating the selector over and over throughout the view.

2. Next, you'll want to attach event handling functions in your view.

```
this.listenTo(this.model, "change", this.render);
this.listenTo(this.model, "destroy", this.remove);
```

You have a choice whether to wire up your event listeners using `model.on()` or `this.listenTo()` ("this" is the view). Although you'll see lots of examples that use `model.on()` I suggest using `this.listenTo()` because it was introduced in later versions of Backbone to help developers fall into the pit of success when using Backbone and not create what most people refer to as "Zombie" views that hang around and eat up memory and cause bugs. This works because Backbone version ~0.9.9 and higher Backbone.Views have a remove method that calls `.stopListening()` which detaches any events that were registered using `.listenTo().` This gets called in the example because of this line:

```
this.listenTo(this.model, "destroy", this.remove);
```

To better understand how a "Zombie" view gets created consider that views are commonly used and then thrown away while models tend to stick around for longer periods of time in a single-page application. The model has a direct reference to a function on the view so even if we replace the original view variable with a reference to a new view, the original view will not fall out of scope because it is still referenced by the model.

3. If the view is a collection view then you'll frequently call fetch on the collection setting "reset" to true which causes all add, change, and update events to not fire while the collection is wholesale replaced from the server. One single "reset" event is fired which is much more efficient than annadd, update, or change event firing for each item in the list. More specifically, an event handler is written to handle the "reset" event on the model and render/re-render each item in the collection. We can see an example of this in projects-view.js which we'll look at next but here are the relevant lines from the view:

```
app.ProjectsView = Backbone.View.extend({
    initialize: function () {
        _.bindAll(this, "render", "addOneProject", "addAllProjects");
        this.listenTo(app.projects, "reset", this.addAllProjects);
app.projects.fetch({ reset: true });
```

```
    },
});
```

Note the fetch must come after the event listener so that the initial render of the page will work properly.

You'll also see numerous examples that fetch the collection data in the router and pass it into the view in the initialize (constructor) function of the view. With Backbone there is no "right way" but these are the patterns you'll see.

The "render" View Function

The render method has its own pattern or common implementation as well.

```
app.ProjectView = Backbone.View.extend({
    template: _.template($("#project-template").html()),
    initialize: function () {
    },
    render: function () {
        this.$el.append(this.template(this.model.toJSON()));
        return this;
    }
});
```

1) Inject the template (already compiled in the template property) with the model's data by calling .toJSON() on the model to get just the properties and leave all the Backbone.Model's change tracking and behavior out.

2) Take the resulting HTML fragment and either .append() or .html() it unto an element using jQuery. .append() is used when render is looping through a collection and .html() is used with there is only one HTML fragment.

Note: these first two steps are commonly done in one line as follows:

```
this.$el.append(this.template(this.model.toJSON()));
```

3) Select all your DOM elements using jQuery. It is a common convention in backbone applications to have variables that reference DOM elements wrapped in jQuery wrapped element sets pre-fixed with $. For example, the example selects the inputs into $name and $description. Note that sometimes you can get the references to the DOM elements in initialize but often this is too early because the elements haven't been rendered if they are part of the template you are rendering.

```
this.$name = this.$(".name");
this.$description = this.$(".description");
```

4) Return the view by returning "this"

This is done because it allows for a more fluent API with method chaining where you can do this in the parent view that renders the child view for each item (project in this case):

Do this:

```
var view = new app.ProjectView({ model: project });
this.$projectList.append(view.render().el);
```

..instead of this

```
var view = new app.ProjectView({ model: project });
view.render();
this.$projectList.append(view.el);
```

"other" Event Handling View Functions

At this point I've discussed most of the code in project-view.js except the custom functions written to handle user interface events like button or link clicks. As I mentioned before, these events were attached in the "events" property of the view.

```
events:{
    "click .cancel":"cancel",
    "click  .project-save":"save",
    "click .remove":"clear"
}
```

We'll look at each of these more closely but for now let's turn our attention to the the project list view.

Projects List View

The project list view (projects-list.js) manages the <ul> that contains the list of projects as well several child views including one that manages adding a new project and several instances of a view that manages each project item in the list.

```
// js/views/projects-view.js

app.ProjectsView = Backbone.View.extend({
    template: _.template($("#projects-template").html()),
    el: "#content",
    initialize: function () {
        _.bindAll(this, "render", "addOneProject", "addAllProjects");
        this.listenTo(app.projects, "reset", this.addAllProjects);
        this.listenTo(app.projects, "add", this.addOneProject);
        this.render();
        app.projects.fetch({ reset: true });
    },
    render: function () {
        this.$el.html(this.template());
        this.$projectList = this.$("#projects");
        var addProjectView = new app.ProjectAddView({
        model: new app.Project(), collection: app.projects });
        addProjectView.render();

        return this;
    },
    addOneProject: function (project) {
        var view = new app.ProjectView({ model: project });
        this.$projectList.append(view.render().el);
    },
    addAllProjects: function () {
        this.$projectList.html('')
        app.projects.each(this.addOneProject, this);
    }
});
```

This view is a collection view and does most of the applicable steps outlined earlier but let's look at the el property and initialize and render functions because there are a couple unusual things going on with this view.

The first thing to notice is the "el" property for this view is actually the main content div in the

"shell" index page: #content. In some examples I've noticed this type of view is named AppView because it manages more than just the list (<ul>) of projects. The view is also responsible for managing the form where users add new projects. You'll see examples of people setting the "el" property in the view definition as is done here as well as examples where the "el" is not set until the view object instance is created.

So this:

```
projectsView = new app.ProjectsView();
```

Versus this

```
projectsView = new app.ProjectsView({ el: '#content' });
```

The second approach has the advantage of allowing you to use the same view in multiple places in your application. We went with the first approach because the view has a default el but that el can be overridden using the second approach.

The initialize follows step 1 binding all functions to "this" (the view) , then step 2 binding the collection events, and also step 3 to fetch and reset the collection data. The one unusual line is the call to this.render before fetching the data. This is necessary because the <ul> that is being appended to when view items are added is not available until after the view template has been rendered. This line could be avoided by putting the <ul> in the page and not in the template. However, when the entire view is replaced (which happens when we load the lists-view.js of todo lists associated to the project in the next section) those original page elements would have been wiped out when we return to the list view.

Immediately Invoked Function Expressions

The more you code in JavaScript the more you'll notice something that has become known as an immediately invoked function expression (IIFE). An IIFE means you create a function and then you immediately call it. You generally create an IIFE because of privacy or more specifically you are trying to keep the implementation details of something private. The reasons for immediately calling it come down to either 1) you only need to call it once or 2) you don't actually return anything you need from the function ...but you want to keep the code private and out of the global namespace.

Imagine you have a function:

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

To make it an IIFE you wrap it in parenthesis and then call it immediately by adding two parenthesis to the end as follows:

```
(function () {

})();
```

The most common use of this pattern is to avoid conflicts in pages and plugins between jQuery and other libraries that use $. To augment our IIFE to do this we simply have our function take an argument of $ and when we immediately call our function we pass "jQuery" instead of the short-hand $ but it is renamed back to $ in our function signature and consequently inside our function and isolated from other conflicts.

```
(function ($) {
    //do some DOM manipulation here with $ and be confident it is jQuery
    //because we are in the closure of the IIFE we wrote
    //$("selector").hide();
})(jQuery);
```

Wrap all Views in IIFE

You'll notice we wrap all the views in the example in a IIFE that takes one argument or import of jQuery which is then aliased to the familiar $ within the view. This pattern of wrapping each view in an IIFE that imports jQuery is a best practice. Backbone.Views use jQuery or another DOM manipulation library because their whole purpose is to manage a root DOM node and it's children so this is a great way to ensure which DOM manipulation library your views are using.

The last thing we need to do to see data on the page is to instantiate this view so we can start using it so we'll turn out attention to the Backbone.Router in the next section.

Application Router

The Backbone.Router maps URLs to functions. In the functions we simply create our views or trigger events for an existing view to respond to by changing itself.

```
var AppRouter = Backbone.Router.extend({
    routes: {
        "": "showProjects",
        "projects": "showProjects",
        "project/add": "addingProject",
        "project/edit/:id": "editingProject",
    },
    showProjects: function () {
        projectsView = new app.ProjectsView();
    },
    addingProject: function () {
        app.projects.trigger('adding:project');
    },
    editingProject: function (id) {
        app.projects.trigger('editing:project', { id: id });
    }
});

app.appRouter = new AppRouter();
Backbone.history.start();
```

Project Wrap-up

At this point you should be able to add, edit, delete and view projects with the code you've written or by using the example project. In the next section, I'm going to discuss the rest of the code at a much higher level so you can get a feel for where you might run into challenges as your application gets more complex.

The first routes and its corresponding function are all we need to get the projects view showing.

```
routes:{
    "":"showProjects",
      "projects":"showProjects",

      },
showProjects:function(){
    projectsView = new app.ProjectsView();
},
```

Note that in order to get the router working you'll need to not just define it but create an instance of it and call `Backbone.history.start()` which tells Backbone to start watching the browser's URL and respond when it changes using the router.

Alternatively, you'll see lots of examples that simply instantiate an instance of a view in a `jQuery $(document).ready`. This would work here but would quickly have to be switched out for a router as shown above so we started with the router.

Check our work

At this point we can check our work so far by adding a project in the browser's console. Setup the backbone/projectmanage folder as a website using your favorite web server. When the page loads you should see a "Projects" header. Open your browser's JavaScript console and run the following line:

```
window.app.projects.create({ name: 'first project' }, { wait: true });
```

You should see an item added to the list. If you have problems you can get the source code for the entire project by following the directions in the Chapter "Example Application Overview" in the section "Setting up the Samples."

Project View Event Handling

Functions

So now that we have a project added let's talk about the "other" event handling view functions in the Project View.

Editing

The editing event handler is triggered when a user clicks the edit link and the URL changes to a route we defined in the router. The router responds by raising an `"editing:project"` event which we are listening for in each item view and respond to by running the "editing" event handler function below.

```
editing:function(args){
    if(args.id!=this.model.id){
        return;
    }
    this.$el.addClass("editing-project");
    this.$name.focus();
}
```

The args come from the URL and are passed by the Router. Since the edit event will be fired on each project view in the list we need to check to make sure the edit event is for this particular project. Next we simply add a CSS class which sets display: block to show the div with the form to edit the project. Lastly, we use this.$name variable we set to the name input element using a jQuery selector in the view's render to set the focus on the first element in the form.

When the form shows for a given project we also show a "cancel" link. The event handler function is implemented as follows.

```
cancel:function(e){
    e.preventDefault();
    this.$el.removeClass("editing-project");
    app.appRouter.navigate('');
}
```

It is important to prevent the default event of navigating to the links location so the back button continues to work as desired. Also, the `router.navigate()` method simply changes the URL and does not actually trigger the route code to run unless you call it as follows: `app.appRouter.navigate('', {trigger:true}); ...`in this case we don't need the full view to render again since we already put the view back into the correct state by removing the CSS class "editing-project."

Deleting

The clear event handler is triggered when a user clicks the remove link and the "clear" event handler function is run because it was defined in the events property of the view.

```
events:{
    "click .remove":"clear"
}

clear:function(){
    this.model.destroy();
}
```

Notice how we chose not to change the URL for this event because it is not an event we want repeated. Deleting the model is as simple as calling destroy on the model for the project list item. Also, remember we keep this view from being turned into a "Zombie" by wiring the destroy event on the model to the remove event on the view.

```
this.listenTo(this.model, "destroy", this.remove);
```

Backbone.Model's destroy method removes the items from the collection and immediately sends a delete call to the RESTful API on your server. Since our models are currently using localStorage .destroy will immediately delete the key, and value entry from localStorage. You may be tempted to name the event handler on your view "remove" instead of "clear" but Backbone.View already has a remove method that destroys all views after removing their event handlers from the model. So don't name your event handler "remove" instead use a close synonym such as "destroy" or "clear".

Project Add View

Remember this view is a child view of the project list view and is the "add project" button and corresponding add form that gets toggled into view using the button.

```
// js/views/project-add-view.js

app.ProjectAddView = Backbone.View.extend({
    template: _.template($("#project-add-template").html()),
    el: "#add-project",
    events: {
      "click .cancel": "cancel",
        "click .project-save": "save"
    },
    initialize: function () {
        _.bindAll(this, "render", "adding", "cancel", "save", "clearFields");
        this.listenTo(this.model, "change", this.render);
        this.listenTo(app.projects, "adding:project", this.adding);
    },
    render: function () {
        this.$el.html(this.template(this.model.toJSON()));
        this.$add = this.$(".add");
        this.$name = this.$(".name");
        this.$description = this.$(".description");
        return this;
    },
    adding: function () {
        this.$el.addClass("adding");
        this.$name.focus();
    },
    cancel: function () {
        this.$el.removeClass("adding");
    },
    save: function (e) {
        e.preventDefault();

        var newProject = new app.Project({ name: this.$name.val(),
            description: this.$description.val() });
        if (!newProject.isValid()) {
            alert(newProject.validationError);
            return;
        }
        this.collection.create(newProject, { wait: true });
        this.$el.removeClass("adding");
        this.clearFields();
        this.$add.focus();
```

```
    },
    clearFields: function () {
        this.$name.val('');
        this.$description.val('');
    }

});
```

Below is a code snippet from the projects list view showing where this view gets added.

```
render:function(){
    //code omitted for brevity here

    var addProjectView = new app.ProjectAddView({
    model:new app.Project(),collection:app.projects});
    addProjectView.render();

    return this;
}
```

In general, this view follows the common steps I outlined for the initialize and render methods. The add view also has several custom event handlers including "adding", "cancel", "save" which are very similar to the custom event handlers in the project view. So at this point I would encourage you to review the sections on the common properties and functions of a view if anything doesn't make sense. As you do, you'll notice the repetitive nature of the code and the common steps we described and begin to see how common these patterns are in Backbone Views.

There is one nuance related to creating a new project that should be explained. When you add a new project in the save function of the view you'll get an error when the new project item view gets rendered because there will be no "id" property for the project you just created and the view needs one to create the edit link (/edit/[id]). This happens because the method commonly used to create a new model collection.create has a default implementation of adding the model to the in memory collection first and then making the remote API call (which is actually persisting to localStorage in our example) which generates the model's id. As soon as the model is added to the collection an "add" event is fired but the

model does not have an "id" property yet. The fix for this is simply to pass an additional option to collection.create which tells it to "wait" until the model is persisted (and has an id) before adding it to the collection.

```
this.collection.create(newProject, { wait: true });
```

The other item that deserves a bit more explanation is the model validation in the view's save function. We already explained how to implement the validation rule but now we see how calling model.isValid() returns a boolean value and sets the model.validationError property with a string describing the validation error. In our example, we simply alert the message but you can easily imagine how a div on the page could be set to the message and styled to create a better user experience.

Challenges

In this section I'm going to discuss the challenges I ran into as the example application became complex. I added another page with multiple "todo" lists for a given project as well as multiple "todos" for each of those lists.

Now that you have a basic understanding of the framework, I will be discussing things at a much higher level. The goal is to give you a feel for what issues you will face as your application grows.

Hierarchy

On the new pages I had a root project model with a child collection of todo lists. Furthermore, each todo list had a child collection of todo item models. The relationships look like this:

Project
  Lists
    Todos

This created the need for parent and child views and model structures. Since Backbone.Model's architecture really only accounts for a collection of models but not hierarchy within those models I needed to figure out how to manage these relationships manually.

Author's Note

There is a plugin Backbone Relational that would have helped a lot with this problem but I've tried my best to use the basic frameworks without plugins in my examples to keep a level playing field between the frameworks.

Model Changes

I added a parent key to child model object. For example, list.js now has a projectid property. I also added a child model collection to the child view and loaded it in the initialize function. See the todos collection below.

```
app.List = Backbone.Model.extend({
    defaults: {
        projectid: '',
        name: '',
        description: '',
        todos: new app.Todos()
    },
});
```

Next, I filtered the collection to only children of the parent using collection.where.
After the child collection is persisted and fetched again it is returned as an array and not a Backbone.Collection so it needed to be wrapped in a Backbone.Collection again. For more information see this Stackoverflow question.

Another challenge I ran into was efficiently loading the todos from local storage without loading all of them. How can I get Backbone.Collection.fetch to not return the entire database table for a given entity or model type. The answer is by passing parameters such as page numbers or filters to the RESTful service. Unfortunately, the localStorage plugin does not allow you to query out of it without bringing all the data into memory. Admittedly, as this example got more complex it was a bit of stretch to use HTML5 local storage as persistence for the application. I worked around this by keeping an app.todos collection of all todos and querying that in-memory collection to get the subset of child items applicable to a list.

Templates

I added a placeholder element to the parent template for child lists. I then loaded all todos in the render function so parent templates like "list-template" are rendered and existing DOM elements can be selected and have children appended.

```
app.List = Backbone.Model.extend({
    defaults: {
        projectid: '',
        name: '',
        description: '',
        todos: new app.Todos()
    },
});
```

Lack of Opinion

Many find Backbone's lack of opinion about how to use the framework liberating. Others find it disorienting and difficult to be productive without architectural guidance. Here are a few of the questions I struggled with when building the example application.

1. Should you load the data in the view's initialize or in the router and then pass it to the view?
2. How many views should you break your page into?
3. Should you call render on the view from the router or have initialize call render on your behalf?
4. Should the view attach to an existing page element as part of it's implementation or should it be appended to the page in the router and/or passed an el to manage?

It would be simpler if there were clear answers to these questions but there are not and the answers will change based on your situation so my best advice is to embrace the freedom.

Memory Management

Given the lack of architectural guidance, "Zombie" views were easy to create and difficult to track down and properly unwire from all relevant model and DOM events. Finding a solution was difficult because the recommended best practices changed when Backbone introduced the Backbone.View.remove method in later versions of the framework to help with this problem.

Router

It is easy to build an application without good URL and back button support with Backbone. In fact, by following many of the examples available it is quite easy to *not* fall into the pit of success. I may have done more routes than necessary in our example application ("edit" doesn't really need its own route). Most routes were done as part of a refactoring of the code after the application was entirely working without routes. I believe this was mostly due to the fact that the router seems to be an afterthought in most examples.

Container, Root Element, and Template

There are three DOM elements you need to be concerned about when managing a Backbone.View:

1. The container that you will inject your HTML into
2. The template which contains your HTML
3. The el or root element of your view

It is possible for 0, 1, 2 or 3 of these items to be the same element depending on how you write your code which is extremely confusing. I spend a great deal of time figuring this out with Backbone applications and I still don't think I have it right.

Other Small Stuff

Here is a quick list of some smaller issues I experienced which should help as you build more complicated Backbone applications particularly involving hierarchy.

- Buttons and links firing too many events because selector not specific enough. For example, "click .save" versus "click .todo-save" and "click .list-save"
- When listening for "add" event in parent collection view calling AddAll when I only needed to call AddOne
- Still very much like jQuery so it is important to name ids and classes carefully. For example, adding suffixes like "-container" or "-template"
- I continually ran into problems forgetting to use the .get and .set properties on the model.
- Accessing .id instead of .get("id") when matching parents with children
- Although, I am embarrassed to admit it I even got creative and did this:
    - Wrong: this.model.set("projectid") = args.projectid;
    - Correct: this.model.set("projectid",args.projectid);
- When editing the CSS class on parent project it caused .editing style on "todos" nested inside to fire. The fix was to break the .editing style into two styles .editing-project and .editing-todo

CHAPTER 7

ANGULARJS EXAMPLE APPLICATION

Basics

AngularJS can seem daunting at first because of the number of concepts and their unfamiliar names. The reality is that only a few of the concepts are important to understanding the framework and once you understand the vocabulary things quickly make sense. Let's review the steps involved in a view being shown to clear things up.

Step 1: A URL or route is requested so the user can see a view

Router

The request is handled by the AngularJS router known as ng-route ($routerProvider in code) which associates the URL to a controller and a template.

```
app.config(function ($routeProvider) {
    $routeProvider
      .when('/', {
          controller: 'ProjectListController',
          templateUrl: '/templates/projectlist.html'
      })
      .otherwise({
          redirectTo: '/'
      })
});
```

Templates

Similar to other frameworks, there is a default index.html page which serves as a shell for other views. The default page has a placeholder element "ng-view" whose contents are replaced with a template. As always, a template is just a fragment of HTML.

Step 2: A controller gets data and sets it on the view's model

Controllers

The controller is a function itself that takes the view's model (ViewModel) as an argument.

The ViewModel in AngularJS is named $scope.

```
app.controller('ProjectListController', function ($scope, ProjectService) {
    $scope.projects = ProjectService.get();
});
```

The $scope can have other JavaScript functions whose job it is to respond to view events such as a button or link click and change the ViewModel's state appropriately.

```
app.controller('ProjectAddController', function ($scope, $location,
ProjectService) {
    var projects = $scope.projects = ProjectService.get();
    $scope.addingProjectNow = true;
    focus('focusMeEvent');

    $scope.add = function () {
        var project = ProjectService.add({ name: $scope.newproject.name,
        description: $scope.newproject.description });
        projects.push(project);
        $location.path('/');
    };

});
```

Because the controller is not just an object but a function, you'll notice that there is only one main "action" per controller which is unlike server-side MVC frameworks including Rails, ASP.NET MVC etc. (where you'll see a Controller with several action methods like List, Create, Details, Edit, and Delete). The other way controllers in AngularJS differ from server-side MVC frameworks is the fact that they have functions to handle view events after the page initially renders.

MVC or MVVM?

If you've done Rich Internet Application (RIA) development with Flex, Silverlight or WPF, you may have thought to yourself that AngularJS's controller functions to handle view events after the view renders are similar to ViewModels in MVVM frameworks. Although similar to

MVVM, it's worth noting that AngularJS is not MVVM. In fact, AngularJS is careful to call itself a Model-View-Star (MV*) framework because it does differ from traditional MVC and MVVM patterns. I think the important thing is to understand how AngularJS and each of these frameworks work and not try to put them in any pre-existing "pattern box."

Step 3: The view is shown with the data

Data binding

You'll notice there is no call to render the view and pass it a model at the end of the controller's function. One might expect something like this:

return View(viewModel);

This line is not needed in AngularJS because of the two-way data binding going on (it's implicit). As you set data onto the ViewModel ($scope) it is automatically bound to the view and updated so there is no need to make an explicit call. To see this, set a break point in one of the examples where you set multiple properties on the view model right after the first property you set in the controller, then look at the page. You'll see that the data has already been updated in the page.

Any property or function you add to $scope is available for you to data bind to in the view. Scope is the root JavaScript object that holds the data (strings, numbers, and other more complex JavaScript objects) you want to see in your view. In other MVC/MV* frameworks it is the model or more specifically the ViewModel.

Services

How did the controller get the data? This is where AngularJS services come into play. You could either use a pre-built AngularJS service like $http which just wraps jQuery's $.ajax() to make an AJAX call to get your data or build a custom repository object to keep all the data

access logic encapsulated and re-usable as I did in the example.

Services are essentially any custom objects you write that you want to work with AngularJS's dependency injection capabilities. Working well with AngularJS's dependency injection makes it easy to reuse the object, unit test your code, and switch out implementations of the object.

```
//injecting a service into a controller and calling it
app.controller('ProjectListController', function ($scope, ProjectService) {
    $scope.projects = ProjectService.get();
});

//service definition
app.factory('ProjectService', function () {
    var projects = [];
    return {
        get: function () {
            var currentProjects = angular.extend([], projects);
            return currentProjects;
        },
    };
});
```

Dependency Injection

For those not familiar with dependency injection, it can be simplified as follows:

1. Do not use the "new" keyword to create an object when another object needs it
2. Instead pass the needed object to the object that needs it (depends on it) as a constructor argument (in some use cases the object is set as a property instead of passed to the constructor)

This seemingly unimportant nuance in how objects get other objects they need is important because it creates a loose coupling between them so that one implementation of a dependency can be substituted for another at testing time.

AngularJS has a simple dependency injection implementation that uses object names.

Basically, it just looks at the argument to the constructor function and looks for an object of the same name and then creates an instance of it and passes it to the function for you. In the example ProjectListController shown above, a ProjectService object is created and made available for use in the controller.

Modules

Lastly, all of these objects: routers, controllers and services need to be organized and not conflict with other code so AngularJS has modules which are essentially like namespaces or packages. So you do something like this to establish the namespace:

```
var app = angular.module('projectmanage', []);
```

And then:

```
app.controller('ProjectListController',function($scope, ProjectService){ ...
    app.factory('ProjectService',function(){ ...
        app.config(function($routeProvider){...
```

We'll talk about these basic building blocks in more detail later, but that is pretty much all there is to AngularJS so let's dive into the example application.

Static HTML

We will have a single index.html page in our application whose contents will be replaced by the files in the templates directory. The router associates a template with a controller as we'll see in the next section but the templateUrl property can also be set to a string identifier instead of a path to a template and will then look for the template in the index.html page in a script tag with that id similar to how templates worked in Backbone (but with convention and not code). The example below (shown earlier) demonstrates that templates can be in the same file:

```
// html
<script type="text/ng-template" id="embedded.home.html">
    <h1> Home </h1>
    {{greeting}}
</script>
<script type="text/ng-template" id="embedded.about.html">
    <h1> About </h1>
    {{content}}
</script>
<div>
    <div id="navigation">
        <a href="#/home">Home</a>
        <a href="#/about">About</a>
    </div>

    <div ng-view></div>
</div>
// javascript
var templatesExample = angular.module('FunnyAnt.Examples.Templates', []);
      templatesExample.controller('HomeController', function ($scope) {
     $scope.greeting = "Welcome to the application.";
});
templatesExample.controller('AboutController', function ($scope) {
    $scope.content = "As a software developer, I've always loved to build
                      things...";
});

templatesExample.config(function ($routeProvider) {
    $routeProvider.
    when('/home', {
        templateUrl: 'embedded.home.html',
        controller: 'HomeController'
    }).
```

```
    when('/about', {
        templateUrl: 'embedded.about.html',
        controller: 'AboutController'
    }).
    otherwise({
        redirectTo: '/home'
    });
});
```

The preferred way of handling templates in larger scale applications where performance is a concern is to concatenate and register your AngularJS templates in the Angular $templateCache at compile time with a build task such as this one.

Installation

Installation of AngularJS v1.1.5 used in this example requires you to download the file from AngularJS.org and add a script tag to the bottom of the page as shown. Again, the scripts are at the end of the page before the body tag so they don't block the page from loading.

```
// index.html

<!DOCTYPE html>
<html ng-app="projectmanage">

<head>
    <meta charset="utf-8" />
    <title>Project Manage</title>
      <link rel="stylesheet" href="js/vendor/jquery-ui-
      1.10.4.custom/css/smoothness/jquery-ui-1.10.4.custom.min.css" />
    <link rel="stylesheet" href="css/style.css" />
</head>

<body>

    <ng-view></ng-view>


    <script src="js/vendor/jquery-1.9.1.min.js"></script>
    <script src="js/vendor/angular.min.js"></script>

</body>

</html>
```

I've also included jQuery but AngularJS does have a slimmed down version of jQuery included with the framework named jqLite so this line is not necessary. But including jQuery will make our lives easier as the application becomes more complex, particularly when writing directives which is where all DOM manipulation should take place in an AngularJS application. If you include jQuery's script tag before AngularJS's as I did here, the full copy of jQuery overrides jqLite and becomes available without you needing to write any other code. I have also included a style.css sheet for our use as needed.

Note: The example AngularJS application uses the v1.1.5 and the built-in router ng-route. As of version 1.2.0 the ng-route is no longer included in the core framework. So to get the example working you would need to pull it down as a separate script file or use angular-ui router which comes as separate file and will have some minor differences.

Templates

The project list HTML template is shown below and will be the template for all the project related functionality in the application. Note that there is not a one to one relationship between templates and controllers and in this case the project list template interacts with four different controllers. Controllers can be defined declaratively using an ng-controller attribute on a HTML tag but in our example application they are associated to controllers through the router which we'll see next.

```
// templates/projectlist.html

<div class="sheet">

    <h2>Projects <a class="action" ng-hide="addingProjectNow"
      href="#projects/add">Add a project</a></h2>

    <form ng-submit="add()" class="inline" ng-show="addingProjectNow">
        <input ng-model="newproject.name" auto-focus
            placeholder="Give the project a name">
```

```
        <input ng-model="newproject.description"
            placeholder="Enter a description">
        <button>Save</button> or <a href="#/">cancel</a>
    </form>

    <hr />
    <ul class="list">
        <li ng-repeat="project in projects" ng-class="{editingproject:
             project == editedProject}">
            <span class="project-display">
                <span class="action-bar">
                    <a href="#projects/remove/{{project.id}}"
                        class="action">delete</a> 
                    <a href="#projects/edit/{{project.id}}"
                        class="action">edit</a>
                </span>
                <a href="#projects/{{project.id}}/lists">{{project.name}}</a>
            </span>

            <form ng-submit="doneEditing(project)" class="inline project-
      form">
                <input ng-model="project.name" auto-focus>
                <input ng-model="project.description">
                <button>Save</button> or
                <a href="" ng-click="revertEditing(project)">cancel</a>
            </form>

        </li>
    </ul>

</div>
```

You'll also notice lots of AngularJS directives in the markup. In my examples, they all start with ng- . If you desire your markup to pass HTML validation you can simply prefix them with data-ng. We'll go deeper into specific directives as they are used in the application.

Application Router

The router is very similar to the Backbone router but instead of associating a route (URL) with a function, AngularJS associates a route to an entire Controller. This can be limiting, as we'll discuss, but also results in very small, single-purpose controller definitions that do one thing well.

```
// js/app.js

var app = angular.module('projectmanage', []);

app.config(function ($routeProvider) {
    $routeProvider
      .when('/', {
          controller: 'ProjectListController',
          templateUrl: '/templates/projectlist.html'
      })
      .when('/projects/add', {
          controller: 'ProjectAddController',
          templateUrl: '/templates/projectlist.html'
      })
      .when('/projects/edit/:id', {
          controller: 'ProjectEditController',
          templateUrl: '/templates/projectlist.html'
      })
      .when('/projects/remove/:id', {
          controller: 'ProjectRemoveController',
          templateUrl: '/templates/projectlist.html'
      })
      .otherwise({
          redirectTo: '/'
      })
});
```

Modules

Before we get into the router code AngularJS wants us to organize our code instead of just placing all our code in the global namespace (on the window object). More specifically, we need to create a module for our router to live inside.

A module is really just an object that serves as a namespace for all the related code. A module for .NET developers is analogous to a namespace, for Java developers it's a package, and for Ruby developers the module terminology should be familiar. Most developers take one of two directions with their modules. Some developers use them to group all the different kinds of objects (controllers, directives, services) involved in a functional area of their application (i.e. project management or invoicing, etc...). Other developers group all similar objects together (myapp.controllers, myapp.directives). Modules are also used to scope your declarative HTML which is why you'll notice at the top of the index.html page an ng-app directive. The ng-app directive refers to a module defined by a call to `angular.module()` as show in the example below.

```
ng-app initializes a module in AngularJS as follows:
    var app = angular.module('myApp', []);
```

Then the module is used in HTML to tell AngularJS which parts of the page it is responsible for managing. In this case it's the entire page but it could be a much smaller area.

```
<html ng-app="myApp">
```

You should also create controllers and other objects inside a module:

```
var app = angular.module('myApp.Controllers', []);

app.controller('ProjectController', function ($scope) {
});
```

At the end of the day it's all about scoping your code so it will play well with other code. More

specifically if you run the example application and type the following in your Chrome browser's console:

```
keys(window)
```

You'll see that all the code is scoped under a single variable named app (I would suggest changing that variable name in your production applications but I chose to leave it because it makes the examples easier to understand).

So at the top of the router code you'll see this line which creates the module.

```
var app = angular.module('projectmanage', []);
```

And then you'll see it used throughout the application for all other objects:

```
app.controller('ProjectListController',function($scope, ProjectService){ ...

app.factory('ProjectService',function(){ ...

app.config(function($routeProvider){...
```

Lastly, you'll see it at the top of the index.html page in a directive so that AngularJS knows how to scope all the objects in your templates.

```
<html ng-app="projectmanage">
```

> Note that it is a very common mistake to forget the ng-app directive and wonder why your application is not working. If you haven't already go add it to index.html page.

Any variable you see in an AngularJS app with a $ prefix like $routeProvider is part of the AngularJS framework. If you create your own custom objects with a $ prefix you are much more likely to conflict with the framework objects so it is recommended to not use the $

prefix when naming your own objects.

Project Service

Here is the code for the project service which is essentially a data access layer or repository in the example.

```
// js/services/projectservice.js

app.factory('ProjectService', function () {
    var projects = [];
    return {
        get: function () {
            var currentProjects = angular.extend([], projects);
            return currentProjects;
        },
        find: function (id) {
            var returnValue = null;
            projects.forEach(function (project) {
                if (project.id == id) {
                    returnValue = project;
                }
            });
            return returnValue;
        },
        add: function (project) {
            project.id = projects.length + 1;
            projects.push(project);
            return project;
        },
        remove: function (project) {
            var index = projects.indexOf(project);
            if (index >= 0) {
                projects.splice(index, 1);
            }
            if (!projects.length) {
                projects = [];
            }
        }
    };
});
```

It is common to put too much business logic and persistent data in your controllers but controllers only have a short lifetime and are not designed to stick around or keep persistent data. One excellent use of services is to create a repository layer and move your data access

and persistence out of your controller so it can be re-used in other controllers throughout the application. Another benefit of getting this type of code out of your controller is it's easier to mock out and unit test because you can just inject an implementation that holds your data in memory as I did in the ProjectService.

This idea of having an in memory data store in a web application can be difficult to understand initially because web developers are used to and expect the stateless nature of the web. In the case of single-page applications, the page never reloads so the data model objects hang around in memory for the entire lifetime of the application or more specifically until the user closes the web browser.

I won't get too detailed about the ProjectService implementation but essentially it just holds an array of projects as a private variable and objects are added, removed, and selected from the array.

Services in AngularJS are confusing for two reasons.

1. It's a broad term with a broad set of uses.

You probably need a service when you need to organize and share code across your application. Common uses of services include data access and persistence, business model objects, and workflow components. But services can also be very user interface specific, for example the AngularJS documentation shows a simple example of a notification service that is coupled to the window object.

2. There are three ways to create and register your own service in AngularJS: service, factory, and provider.

There are differences in the implementations but developers get confused because factories and providers are specific types of services but they are still called services and there is a service way of creating a service (no that was not a typo).

Let me give a quick explanation of the difference between service, provider and factory.

Both .service() and .factory() return an object called a service. The first argument to .service() and .factory() is the name of the service. The second argument to .service() and .factory() is a function that is used to help initialize the service object.

With .service() the function is a JavaScript constructor function which is called by AngularJS with the JavaScript "new" keyword so there is no need to create a new object that will be the service object that is returned from the function. Instead, JavaScript returns this automatically from the function and the developer's only responsibility is to initialize the "this" object by creating properties or methods on it.

With .factory the function is closer to what experienced JavaScript developers call the "module design pattern" where you create an object, initialize it by setting properties or methods on it which you wish to be "public" and then return it from the function.

These following two services are functionally equivalent in AngularJS. The developer simply needs to choose which syntax they prefer (most examples use the .factory syntax).

```
app.service('UserService', function () {
    // service is just a constructor function
    // that will be called with 'new'

    var users = [{
        name: 'John Doe',
        value: 1
    }, {
        name: 'Jane Doe',
        value: 2
    }];
    //no need to new up an object
    this.get = function () { return users };
    //or return an object, this is returned
});
```

```
app.factory('UserService', function () {
    // factory returns an object
    // you can run some code before we return anything
    //the code we run before 'return' stays private
    //unless we return it

    var users = [{
        name: 'John Doe',
        value: 1
    }, {
        name: 'Jane Doe',
        //value: 2
    }];

    return {
        get: function () {
            return users;
        }
    };
});
```

So that leaves provider. There are use cases for creating a service as a provider, they are just much less frequent. Providers are the only service you can pass into your .config() function. Use a provider when you want to provide module-wide configuration data to your service object before making it available. This blog post does an excellent job explaining the differences and has a good provider example.

Project List Controller

Below is the controller for showing the list of projects.

```
// js/controllers/projectcontroller.cs

app.controller('ProjectListController', function ($scope, ProjectService) {
    $scope.projects = ProjectService.get();
});
```

The controller takes $scope (the ViewModel as do all controllers) and ProjectService as a dependency that gets injected by AngularJS.

The template's use an ng-repeat directive to render the $scope.projects array of projects set by the controller.

```
<ul class="list">
    <li ng-repeat="project in projects"
      ng-class="{editingproject: project == editedProject}">
        <span class="project-display">

            <span class="action-bar">
                <a href="#projects/remove/{{project.id}}"
                    class="action">delete</a> 
                <a href="#projects/edit/{{project.id}}"
                    class="action">edit</a>
            </span>

            <a href="#projects/{{project.id}}/lists">{{project.name}}</a>

        </span>

        ....code omitted for clarity
    </li>
</ul>
```

Notice how clean the markup is you feel like you're just reading HTML.

There are many different syntaxes for creating a controller as shown below.

```
function($scope, ProjectService){
    $scope.projects = ProjectService.get();
}
```

Simple, but doesn't put the controller inside a module.

```
angular.module('app')
.controller('ProjectListController', ['$scope', 'ProjectService', function
($scope, ProjectService) {
    $scope.projects = ProjectService.get();
}]);
```

Solves the problem where AngularJS breaks when minified because the names of the dependency variables are changed so AngularJS's dependency injection convention of using names to find dependencies no longer works, but is impossible to write and read (too complicated).

```
app.controller('ProjectListController', function ($scope, ProjectService) {
    $scope.projects = ProjectService.get();
});
```

The syntax I used, utilizes modules, is still readable, and I can use ngmin (AngularJS pre-minifier) to solve the dependency injection issues. Basically, ngmin rewrites this cleaner syntax to the more obtuse syntax that works after minification.

More specifically, ngmin turn this:

```
angular.module('whatever')
.controller('MyCtrl', function ($scope, $http) { ... });
```

Into:

```
angular.module('whatever')
.controller('MyCtrl', ['$scope', '$http', function ($scope, $http) {
... }]);
```

Project Add Controller

The ProjectAddController is invoked when the user clicks the add project link:

```
<a class="action" ng-hide="addingProjectNow" href="#projects/add">
Add a project
</a>
```

The link takes them to the "#project/add" route which knows to handle the request with the ProjectAddController function.

```
// js/controllers/projectcontroller.cs

app.controller('ProjectAddController', function ($scope, $location,
ProjectService) {
    var projects = $scope.projects = ProjectService.get();
    $scope.addingProjectNow = true;

    $scope.add = function () {
        var project = ProjectService.add({ name: $scope.newproject.name,
description: $scope.newproject.description });
        projects.push(project);
        $location.path('/');
    };

});
```

The three-way assignment here is just to setup an easy way to access projects on the scope elsewhere in the controller.

```
var projects = $scope.projects = ProjectService.get();
```

In other words, so I can say `projects.push` instead of `$scope.projects.push`.

This variable is used in the template to determine whether to show the "add" form or not.

```
$scope.addingProjectNow = true;
```

The template uses the ng-show directive. Ng-show takes a boolean variable or expression which if true it shows the HTML element.

```
<form ng-submit="add()" class="inline" ng-show="addingProjectNow">
    <input focus-on="focusMeEvent" ng-model="newproject.name"
       placeholder="Give the project a name">
    <input ng-model="newproject.description"
     placeholder="Enter a description">
    <button>
        Save
    </button> or
    <a href="#/">
        cancel
    </a>
</form>
```

What you might start to notice at this point is how easy it is to maintain state in AngularJS without as many element ids and classes. To achieve the hiding or toggling behavior in a traditional server-side web application you would most likely turn to jQuery's toggle method and directly interact with the DOM in code to hide and show elements. Although the hiding and showing in Angular is very declarative because it uses uses attributes like ng-show and ng-hide, the actual "state" about whether you are adding a project is part of the $scope or ViewModel object and the need for many non-semantic element IDS and CSS classes goes away.

The `$scope.add()` function is our first example of an additional function on a controller whose purpose is to respond to a user event. In this case the event is the user clicking the "save" button on the "add" form after it is initially shown.

```
$scope.add = function () {
    var project = ProjectService.add({ name: $scope.newproject.name,
description: $scope.newproject.description });
    projects.push(project);
    $location.path('/');
```

```
};
```

First, we add the project to persistent storage by calling `ProjectService.add()`. Notice how we never declared "newproject" except in the view but its values were automatically bound from the view/template element back to the model. This is one of the many examples where two-way binding results in less code. Note that normally instead of just creating a Javascript object literal on the fly with a name and description as shown I would create a business model object as a service to use. You can see this in the full example with the todo service. Some signs that it is time to create a business object that forced me to do it for the todo but not the project object are when the model gets to be more than just a few properties or the model needs behavior such as building a "status" string up out of two other properties.

Since the ProjectService returns a copy of the in memory project array we need to add it to the ViewModel's ($scope's) project array so that we can immediately see the new object without having to "get" all the projects again.

```
projects.push(project);
```

Lastly, AngularJS has a $location service similar to Backbone's `router.navigate()` that changes the URL. Notice how the $location service is passed as dependency in the constructor function of the controller which makes mocking out this dependency on the web browser easy and unit testing a breeze.

```
$location.path('/');
```

That wraps up the code for the ProjectAddController. Let's move on to editing.

Project Edit Controller

The ProjectEditController is invoked when the user clicks the edit project link in the project list:

```
<a href="#projects/edit/{{project.id}}" class="action">edit</a>
```

The link takes them to the `"#projects/edit/{{project.id}}"` route which knows to handle the request with the ProjectEditController function.

```
// js/controllers/projectcontroller.cs

app.controller('ProjectEditController', function ($scope, $location,
$routeParams, ProjectService) {
    var projects = $scope.projects = ProjectService.get();
    var project = ProjectService.find(parseInt($routeParams.id));
    $scope.originalProject = angular.extend({}, project);
    $scope.editedProject = project;

    $scope.doneEditing = function (project) {
        $scope.editedProject = null;
        //trim
        if (project.name) {
            project.name = project.name.trim();
        }
        //delete blanks
        if (!project.name || project.name.length == 0) {
            ProjectService.remove(project);
        }
        $location.path('/');
    };

    $scope.revertEditing = function (project) {
        projects[projects.indexOf(project)] = $scope.originalProject;
        $scope.doneEditing(project);
        $location.path('/');
    }

});
```

To begin, we load the data including all projects for the list and the project being edited. The

AngularJS service $routeParams is passed in as dependency. The service parses and gives access to URL parameters and query string values as needed in this case the project id. Again, injecting this makes the controller significantly easier to test because I don't have to mock away the entire browser, I just need to send in the parameters needed.

```
var projects = $scope.projects = ProjectService.get();
var project = ProjectService.find(parseInt($routeParams.id));
```

Next, we make a copy of the current project into the "originalProject" variable. This is necessary because the two-way binding happens automatically and immediately so when the user changes the value of the project name that is bound to the project object, the object is immediately changed (on each keystroke) and before the save button is clicked. Now imagine the user wants to "cancel" or undo their changes and clicks the cancel button. We need a copy of the original project to support the undo functionality.

```
$scope.originalProject = angular.extend({}, project);
$scope.editedProject = project;
```

We also set the current project into the "editedProject" $scope variable so it can determine which of the project list items is being edited and then show the project edit form instead of the read-only list item.

```
<li ng-repeat="project in projects" ng-class="{editingproject: project ==
editedProject}">
```

The view uses the ng-class directive which sets a CSS class (in this case "editingproject") on the item if the expression after the colon evaluates to true. This allows the form or the read-only view to be shown depending on whether the "editingproject" CSS class is set. Below are the relevant CSS styles to make this clearer.

```
ul li form {display:none;}
ul li.editingproject form.project-form {display:block;}
```

```
ul li span.project-display {display:block;}
ul li.editingproject span.project-display {display:none;}
```

This is a particularly useful directive if you are good with CSS and need to toggle the visibility of several elements as a group.

The controller has two event handlers for saving the project edits or reverting their changes. You can see them in the markup below:

```
<form ng-submit="doneEditing(project)" class="inline project-form">
    <input ng-model="project.name" auto-focus>
    <input ng-model="project.description">
    <button>Save</button> or
    <a href="" ng-click="revertEditing(project)">cancel</a>
</form>
```

Notice how the relevant project object is available and can be passed back to the controller. Compared to server-side programming where state is constantly forgotten this access to the model without constantly going back to the database is refreshing. Remember this is possible because we are building a fat-client single-page application and "application state" or "models" are kept in memory between different view requests.

The implementation of the "doneEditing" event handler:

```
$scope.doneEditing = function (project) {
    $scope.editedProject = null;
    //trim
    if (project.name) {
        project.name = project.name.trim();
    }
    //delete blanks
    if (!project.name || project.name.length == 0) {
        ProjectService.remove(project);
    }
    $location.path('/');
};
```

First, we set the editedProject to null so that the edit form will no longer be shown. Next, we trim the project name and remove the project if the name is blank.

Since we are working on a project from our repository's project collection (ProjectService) and not a copy, we simply need to update the project object, data binding automatically gets the value from the input and sets it on the project model object at which point it has already been persisted. Depending on how we implemented our repository it might have been necessary to make a call to commit or persist the changes, but it is not in this case.

Lastly, we set the URL back to the default so the change from the form edit mode back to the full list is recorded in the browser history and the back button can be used as expected.

The "revertEditing" event handler is where we take advantage of the "originalProject" project object clone we made in main controller function. We set the view model project collection on $scope named "projects" back to the original project. Then we call "doneEditing" again so the original project can go through the normal steps after an edit to project including hiding the form and setting the URL back to the default for the project list.

```
$scope.revertEditing = function (project) {
    projects[projects.indexOf(project)] = $scope.originalProject;
    $scope.doneEditing(project);
}
```

Project Remove Controller

The `ProjectRemoveController` is invoked when the user clicks the delete link in the project list:

```
<a href="#projects/remove/{{project.id}}" class="action">delete</a>
```

The link takes them to the "#projects/remove/{{project.id}}" route which knows to handle the request with the ProjectRemoveController function.

```
// js/controllers/projectcontroller.cs

app.controller('ProjectRemoveController', function ($scope, $routeParams,
$location, ProjectService) {
    var project = ProjectService.find(parseInt($routeParams.id));
    ProjectService.remove(project);
    $location.path("/");
});
```

The ProjectRemoveController is straightforward. First we find the project using the project id in the URL and the ProjectService, then remove the project via the ProjectService and redirect the URL back to the project list default view. You could imagine improving this by making the view an "are you sure" message and then removing the project using an event handler, but I kept this implementation simple.

Directives

Directives are custom HTML elements, attributes, or classes for your application.

What are Directives?

At a high level, directives are markers on a DOM element (such as an attribute, element name, comment or CSS class) that tell AngularJS's HTML compiler ($compile) to attach a specified behavior to that DOM element or even transform the DOM element and its children.

Angular comes with a set of these directives built in, like ngBind, ngModel, and ngView. Much like you create controllers and services, you can create your own directives for Angular to use. When Angular bootstraps your application, the HTML compiler traverses the DOM matching directives against the DOM elements.

Source: AngularJS Documentation

In Practice

Any time you find yourself wanting to use jQuery to do something in your application you probably want to do it inside a directive. Directives allow you to extend HTML itself and encapsulate your implementation of it.

Auto Focus Directive

A directive we need for our project management example application is one that allows us to set the default focus on an input.

```
app.directive('autoFocus', function ($timeout) {
    return {
        restrict: 'AC',
        link: function (_scope, _element) {
            $timeout(function () {
                _element[0].focus();
            }, 0);
        }
    };
});
```

Directive names should use *camelCase* when you define them. Then you can use either of these syntaxes when using them.

```
<input camel-case /> or <input data-camel-case />
```

The auto focus directive can be used:

```
<input auto-focus /> or <input data-auto-focus />
```

Directives can get pretty complicated and have an obtuse syntax but here are the basics that we need to know to create our auto focus directive:

The restrict option is typically set to:
'A' - only matches attribute name
'E' - only matches element name
'C' - only matches class name
These restrictions can all be combined as needed:

For example, 'AEC' - matches either attribute or element or class name.

There is also an option 'M' which matches comments but it is rarely used or needed.

For our directive we will restrict it to an attribute or a class.

```
restrict: 'AC'
```

Note: If your AngularJS project needs to support IE8 or lower use an AngularJS version <1.3 and do not use custom directives as elements (<foo></foo>) but instead stick to attributes or classes (`<div foo></div>`). IE8 and lower does allow you to use them as elements but requires you to jump through hoops to create custom elements. For more information see the documentation. That said, do not make the mistake of falling for the argument "we can't use AngularJS because we need to support older versions of IE." Simply use AngularJS v1.2.x which is an extremely stable and mature version of AngularJS. All frameworks and applications need to stop supporting older browser versions at some point as they release new versions.

Directives that want to modify the DOM typically use the link option. Link takes a function with the following signature, function link(scope, element, attrs) { ... } where:

scope is an Angular scope object.
element is the jqLite-wrapped element that this directive matches.
attrs is a hash object with key-value pairs of normalized attribute names and their corresponding attribute values

```
link: function(_scope, _element) {
    $timeout(function(){
        _element[0].focus();
    }, 0);
}
```

Our example does not utilize all three arguments to the link function, the attributes hash is not needed. The $timeout service which wraps the browser's Window.setTimeout function is

passed into the directive as dependency. In the end all we do is take the element passed in and set focus on it by calling the jqLite (jQuery in our case because we've included it) focus function on the wrapped element set and we wrap the operation in a $timeout call so that the element can be rendered before setting focus.

We simply set it as a custom attribute on our project name input in the project list template.

```
<input ng-model="newproject.name" auto-focus placeholder="Give the project a name">
```

We are now able to use this for other inputs and did for list names and todo descriptions.

Challenges

As I did previously with Backbone, in this section I'm going to move faster and discuss the challenges I ran into as the example application became complex.

Wrapping plugins and widgets

You are encouraged to limit your interaction with the DOM to directives in AngularJS as well as limit your use of jQuery to the set of functionality in jqLite. In many cases, for example, when I added the modal div on the todo for the user to assign or set the due date, it is quite simple to just set some application state on $scope's collection of todos and determine whether to show the dialog or not using ng-show.

```
//TodoAssignController
var todo = ToDoService.findTodo(parseInt($routeParams.id));
todo.assigningTodo = true;

// templates/todolists.html
<div class="assignment-dialog dialog" ng-show="todo.assigningTodo">
```

But there are some controls that you don't want to re-implement because their functionality is complex to reproduce in a short amount of time. I ran into this when I needed to add a calendar control and decided to used jQuery UI project's datepicker plugin. I could have made my life easier by using the Angular UI Bootstrap project's datepicker but at the time I wrote the example the project did not have a datepicker control and I also tried to imagine being in a legacy application and not being able to include the Bootstrap front-end framework without messing up existing layouts.

The code for the jQuery UI datepicker directive is shown below.

```
app.directive('calendar', function () {
    return {
        restrict: 'A',
        require: 'ngModel',
        link: function (scope, element, attrs, ngModelCtrl) {
            $(function () {
                element.datepicker({
                    dateFormat: 'mm/dd/yy',
                    onSelect: function (date) {
                        scope.$apply(function () {
                            ngModelCtrl.$setViewValue(date);
                        });
                    }
                });
            });
        }
    };
})
```

The code above works but took at least 6 hours of research to get working. Admittedly, this example is particularly complex because the plugin enhances an existing text input with the date picker functionality and the input was already bound to an object using the ng-model directive. Let's get into the details of how it ended up working.

Setting the date without a datepicker

We need to allow the user to set a due date for each todo. To get the basic functionality working we simply need to add an input to the assignment form and two-way bind it to the model using ng-model.

```
<form>
    <label>Assign this to-do to:</label>
    <select ng-model="todo.assigneduser"
      ng-options="user.name for user in users">
        <option value="">Unassigned</option>
    </select>
    <label>Due Date</label>
    <input ng-model="todo.duedate"></input>
</form>
```

Notice that we are adding binding the input to `todo.duedate` not just `duedate` since we want each todo to have its own date.

Adding the datepicker

Since this input requires a date, it would be good to have a datepicker control and as mentioned previously I chose the commonly used jQuery UI project's datepicker.

Since the jQuery datepicker requires significant DOM manipulation to work with, we want to create a custom AngularJS directive for the datepicker that will work as follows:

```
<input ng-model="todo.duedate" calendar></input>
```

Since the jQuery datepicker control sets the value of the input field in code when a user selects a date, AngularJS doesn't know that the value has changed. To be more specific, AngularJS watches for user events on normal input controls to tell if a value has changed and then responds by binding and updating the model but in this case the user isn't interacting with the input field to change the value they are interacting with the pop-up div that displays a calendar. So the directive we write needs to integrate with the AngularJS API for data binding which is a class called the ngModelController and tells the framework when changes have been applied. See the calendar directive code below:

```
app.directive('calendar', function () {
    return {
        restrict: 'A',
        require: 'ngModel',
        link: function (scope, element, attrs, ngModelCtrl) {
            $(function () {
                element.datepicker({
                    dateFormat: 'mm/dd/yy',
                    onSelect: function (date) {
                        scope.$apply(function () {
                            ngModelCtrl.$setViewValue(date);
                        });
                    }
                });
            });
        }
    };
});
```

The `"require ngModel"` property says the element the directive is applied to must also

have a ng-model directive applied to it as well.

So not just:

```
<input calendar></input>
```

But this:

```
<input ng-model="todo.duedate" calendar></input>
```

The parameters to the link function are 1) optional (you can leave them out) 2) fixed in position (for example the first argument always needs to be scope and the fourth the model controller) 3) can be renamed. But be careful what you name your variables, because I found lots of demos and sample code with ngModel as the fourth variable name but if you look up ngModel in the documentation it talks about the ng-model directive itself and not the ngModelController class that is being used in the link function.

Most of the code in the link function is standard jQuery initialization code for the jQuery UI datepicker control see the API reference here.

The onSelect is a jQuery UI datepicker event that provides a hook to run jQuery code when a date is selected in the calendar of the datepicker.

In our directive implementation we need to let AngularJS know the value changed which is done by calling ngModelCtrl.$setViewValue(date).

I found setViewValue to be a misleading and confusing method name because although the implementation of this method in AngularJS is to set all of its internal variables for "application state" one of which is a variable named viewValue, hence the name setViewValue, but it is also updating the model variables which is what we are trying to accomplish. Perhaps a better name for this method might be "refreshBindingState" or "updateBindingState".

The last important thing to notice in this directive is the call to `$scope.apply()`. Apply tells AngularJS that some state changed, so apply those changes or rebind. You can call apply like

this:

```
ngModelCtrl.$setViewValue(date);
scope.apply();
```

But the best practice is to call it like this:

```
scope.$apply(function () {
    ngModelCtrl.$setViewValue(date);
});
```

`scope.$apply()` is really just a wrapper function to call your custom code in a try/catch block and then call `scope.$digest()` which is AngularJS's dirty checking method. Running it inside the apply function is a best practice because if your code throws an error then that error is thrown outside of AngularJS, which means any error handling being used in your application is going to miss it. $apply not only runs your code, but it runs it in a "try/catch" so your error is always caught and the $digest call is in a "finally" clause which means it will be called regardless of the error being thrown.

Setting Focus on previously hidden elements

We previously went into detail about the implementation of the auto-focus directive but this was difficult to get working.

Initially I tried the following solution:

```
app.directive('autoFocus', function ($timeout) {
    return {
        link: function (scope, element, attrs) {
            element[0].focus();
        }
    };
});
```

This works in cases where the input is visible when the page initially loads. But in our case the form with the input is hidden until we click the "add project" link.

So why didn't I just get over myself and use a jQuery selector in the controller and then call focus? Because as soon as we introduce the DOM into the controller then we cannot unit test the controller because it then has a dependency that is not being passed in the constructor, more specifically, an element in the DOM that you want to set focus on.

I tried a very elaborate solution involving a custom service interacting with a directive which was similar to blesh's answer on this stackoverflow question but I wasn't able to get it working on all views in the application.

Eventually I augmented the original solution with the $timeout service inside the link function and things worked. The point here is not the destination (I'm still not excited about the solution) but the fact that it did take significant effort to get this small amount of functionality that involved the DOM because directives can be difficult to master.

```
app.directive('autoFocus', function ($timeout) {
    return {
        restrict: 'AC',
        link: function (_scope, _element) {
            $timeout(function () {
                _element[0].focus();
            }, 0);
        }
    };
});
```

Model

AngularJS does not have a model base object as part of the framework but instead allows you to use plain old JavaScript objects to represent the data in your applications. Although it is liberating for your models to just be plain old JavaScript objects it can also be confusing because the familiar object-oriented concepts like classes and inheritance are missing from JavaScript. In addition, it can be helpful to have a base class for your models which establishes where validation and change tracking should happen. Validation and change

tracking are not missing from AngularJS but can be difficult to find and wrap your head around. Lastly, AngularJS has the concepts of services, factories and providers and it is confusing as to whether these should be used to help represent the business model.

Data access

Angular's data access story can pretty much be summarized as $http is a wrapper around jQuery $.ajax calls. The major advantage of using $http service is that it works well with the framework's automatic two-way data binding. The $resource service is a slightly higher level abstraction that allows you to write less code if your service complies with AngularJS's idea of a RESTful service. At the end of the day, I found myself wanting a more full-fledged object relational mapping solution and/or something that recognizes the rise of real-time communication becoming available in modern browsers.

Router

The basic router functionality that comes with AngularJS 1.x is known as ng-route. It has several limitations that were acknowledged in the Angular team's design document for Angular 2.x and Angular's built in router, ng-route, was actually removed from the framework in version 1.2 and higher. In the Angular team's words: the initial Angular router was designed to handle just a few simple cases. As Angular grew, we've slowly added more features. However, the underlying design is ill-suited to be extended much further. Below are some of the specific issues I encountered.

It was very easy for me to create a first version of my application that ignored URLs and broke the back button. It wasn't easy for me to create an application with deep linking. More importantly, I really needed nested or sibling views. Ideally, I would have had views for each level in my hierarchy (project, list, todo) but that was not possible with ng-route. I could have used the AngularUI project's router in the example application and it would have given me the ability to nest views, but I wanted to keep the examples confined to the core framework.

How many Templates and Controllers

It was difficult to figure out how many templates and controllers were needed for a given view. In the end, I believe I got the controllers correct by making them very granular. Ideally I would have had more small views, but this was not possible without using the AngularUI project's router. I recommend you use the ui-router with your projects.

Conclusion

Despite the challenges, I had a much more productive and rapid application development experience using AngularJS than Backbone as you can see from the amount of code in each of the examples. Next, we'll look at the same application using Ember.

CHAPTER 8

EMBER EXAMPLE APPLICATION

Basics

For someone just learning Ember the framework can be very confusing. This is mostly due to the number of concepts and the names of those concepts. In particular, the route object is different than anything we have seen in other frameworks and different than the router. This leaves the developer wondering what to do in the route versus the controller, or the route versus the router. The framework object names were not chosen to confuse new developers but because the framework has its roots in SproutCore which is a rich client application framework. I will attempt to quickly clear up the confusion by going over each type of object, what kinds of code is in each type of object, and provide a small amount of history so it will be easier for you to remember.

Router

As in other frameworks, Ember's Router maps a URL to some JavaScript code. In the case of Ember, the URL is mapped to a series of nested templates each backed by a model object and the instantiation of the templates and their backing controllers and route objects is done automatically by the framework-- in most cases by following a naming convention.

Ember's router supports nested templates and routes which enables a more robust experience for the developer allowing him to compose his templates from a series of smaller templates. A concrete example of this is a page that shows a list of blog posts and when you click on one the list remains visible on the page but the details of the post you clicked are shown in another region of the page.

Here are some code snippets to help you understand how this can work:

```
App.Router.map(function() {
    this.resource('posts', function() {
        this.resource('post', { path: ':post_id' });
    });
```

```
});
//post details template
<script type="text/x-handlebars" id="post">
    <h1>{{title}}</h1>
    <h2>by {{author.name}} </h2>

    <hr>

    <div class='intro'>
        {{excerpt}}
    </div>

    <div class='below-the-fold'>
        {{body}}
    </div>
</script>


        //posts lists template
        //markup omitted for clarity
<div class="span3">
    <table class='table'>
        <thead>
            <tr><th>Recent Posts</th></tr>
        </thead>
        {{#each model}}
        <tr>
            <td>
                {{#link-to 'post' this}} {{title}} {{/link-to}}
            </td>
        </tr>
        {{/each}}
    </table>
</div>
<div class="span9">
    {{outlet}}
</div>
```

In the above example, a post details template is rendered into the {{outlet}} in the posts list template when a link is clicked in the post.

To make this more clear here is how the router code would change to *not* nest the details template.

```
App.Router.map(function () {
    this.resource('about');
    this.resource('posts', function () {
        //post resource route used to be here
    });
    this.resource('post', { path: ':post_id' });
});
```

We would also remove the `{{outlet}}` from the post list template.

After making this change, when a post in the list is clicked, the entire body of the page would reload with the details template and the list would no longer show.

You may be wondering why the router isn't specifying which controller and template to use. Again, this is because Ember embraces convention over configuration (similar to Ruby on Rails) and has a naming convention that determines which template and controller to use by default and creates these objects on behalf of the developer.

Omitting the Path

Note that you can leave off the path if it is the same as the route name so this:

```
App.Router.map(function () {
    this.resource('projects');
});
```

And this are the same:

```
App.Router.map(function() {
    this.resource('projects', path:{/projects});
    });
```

Route Name

Notice that the first argument to `this.resource()` and `this.route()` is the route name. AngularJS and Backbone don't have a route name so beginners sometimes make the mistake of thinking it is the route path since that argument is optional and can be set by convention. Although having a route name is different than other client-side frameworks, it is similar to how routes are defined in Rails and ASP.NET MVC and brings a similar benefit. With a route name established, links can be created throughout your application based on the route name, not the path, so if the path changes, you only have to make a change in one place. Here is an example link-to block helper which will be explained in the next section in more detail:

```
{{#link-to 'projects'}}cancel{{/link-to}}
```

Resource versus Route

A common source of confusion with the Ember router is understanding when to use this.resource versus this.route in the map function of the router (see the example below). In summary, use resource if the URL represents a noun and route if the URL represents a verb or an adjective.

```
App.Router.map(function () {
    this.resource('projects', function () {
        this.route('create');
    });
});
```

If you're wondering why the framework authors don't just always use route there are a couple of reasons. In practice, route and controller names defined as a resource are prefixed with the

resource name, for example, ProjectsCreateRoute and ProjectsCreateController. Contrastingly, resources stand by themselves and do not have a prefix, for example, ProjectsRoute and ProjectsController (this can help keep your Route and Controller names from becoming too long). In theory, the "resource" term was taken from the nomenclature of Representational state transfer (REST), a software architectural style.

Templates

Handlebars

Ember.js uses the Handlebars templating library to intersperse dynamic content with your HTML. Handlebars uses the popular mustache style bindings to indicate dynamic parts of the page.

```
<p>{{ dynamic_variable }}</p>
```

Similar to AngularJS, these dynamic variables are two-way data bound so your views will be updated whenever model values change.

There are only a few Handlebars logic helpers you will use frequently in Ember applications and they are: `#if, #unless,` and `#each.` They are straightforward but here are some examples of their uses:

Handlebars logic examples

```
//#each
<h1>Comments</h1>

<div id="comments">
    {{#each comments}}
    <h2>{{title}}</h2>
    <div>{{body}}</div>
    {{/each}}
</div>
  //#if
    //You can use the if helper to conditionally render a block. If its
argument returns //false, undefined, null, "" or [] (a "falsy" value),
Handlebars will not render the //block.
<div class="entry">
    {{#if author}}
    <h1>{{firstName}} {{lastName}}</h1>
    {{/if}}
</div>
//unless
//You can use the unless helper as the inverse of the if helper. Its block
will be //rendered if the expression returns a falsy value.
<div class="entry">
    {{#unless license}}
    <h3 class="warning">WARNING: This entry does not have a license!</h3>
    {{/unless}}
</div>
```

Handlebars Ember Extensions

Ember extends Handlebars to create framework specific concepts.

Rendering

Many of these extensions are related to single-page applications and the fact that after an initial page load, part of the page needs to be replaced on future requests.

An outlet is placed on the main or shell page inside the application template to indicate where a new template should be replaced when the URL changes.

{{outlet}}

An outlet placed inside a template other than the default application template will load a nested route.

The block helper `{{#link-to 'routeName'}}` creates an HTML anchor tag or link that when clicked will cause the contents of the {{outlet}} to be replaced by the route's associated template.

The helper `{{render 'routeName'}}` replaces the contents of a region with a template without requiring the URL to change. This is useful when you want to break up a complex template into smaller pieces, for example, if you have a list of projects and want a template for each project in the list. The template for a single project list item would be rendered with {{render}} as part of the initial project list template.

Events (actions)

Another extension to Handlebars you'll frequently use is the {{action}} helper. The action helper is a way to declaratively trigger event handlers on your controller after the template is initially rendered. In our example application the most common use was with a button.

```
<button {{action 'save' }}>Save</button>
App.ProjectsCreateController = Ember.Controller.extend({
    actions: {
        save: function () {
            //save code here
        }
    }
});
```

The default event is "click" but it supports many others and can also be used with other elements as well.

Example 1: on double-click

```
<label {{action "editTodo" on="doubleClick" }}>{{title}}</label>
```

Example 2: on click while holding down 'alt' key

```
<div {{action 'anActionName' allowedkeys="alt" }}>
```

Example 3: on click passing Model object

```
<a href="#" {{action 'cancelEditing' project}}>cancel</a>
```

Forms (Input Helpers)

Working with inputs in Ember does not work the way one might expect. It is not as simple as setting the value equal to a handlebars expression, because that doesn't setup two-way data binding, but only gives you one-way data-binding from the model to the view.

```
<input type="text" value={{firstName}} size="50" /> // doesn't setup two way binding
```

Instead, you can simply use the bind-attr to establish two-way data-binding for each attribute:

```
<input type="checkbox" {{bind-attr disabled=isAdministrator}}>
```

But Ember has specific form or input helpers that make creating form elements easier. The syntax for these helpers is essentially to replace brackets with double curly braces. After doing this, attributes that are 'quoted' will be rendered as is. However, when left unquoted, these values will be bound to a property on the template's current rendering context (usually the controller). For example:

```
{{input type="text" value=firstName disabled=entryNotAllowed size="50"}}
```

The helper shown above will render the text shown below. The value and disabled attributes are dynamically set based on values on the controller (current context) while size is rendered as a string.

```
<input type="text" value=Craig disabled=false size="50" />
```

Checkboxes and textarea have helpers with similar syntax.

```
{{input type="checkbox" name="isAdmin" checked=isAdmin}}
```

```
{{textarea value=name cols="80" rows="6"}}
```

Views

Views in Ember are very similar to Backbone Views and *are code* and not HTML markup. Because Handlebars templates are so powerful the majority of your application's user interface can be built without the use of Ember Views. In fact, our example does not require the use of an Ember View. An Ember View is an object that wraps a template and is sometimes (but infrequently) necessary when you need sophisticated handling of user events and/or to create a re-usable component. Since the example application does not require any I will not go further into them but refer you to the documentation on Ember Views.

Note that it is not uncommon for Ember templates to be referred to as views or view templates by those not familiar with Ember vernacular since they are very similar to AngularJS views.

Components

Components are essentially a reusable template with behavior which can be used in other templates as a custom element. They are an early implementation of the W3C's web components specification for a custom element implementation. Ember components are similar to AngularJS directives in their mission but are generally easier to implement. We did not have a need for a component in our example application, so I will not go into a detailed explanation. Also the implementation of a component is essentially a small template where

the model data is passed in from the current context (Controller or Route) so once you understand the core concepts in Ember they are easy to build.

```
//component
<script type="text/x-handlebars" id="components/blog-post">
  <h1>{{title}}</h1>
  <div class="body">{{yield}}</div>
</script>
```

```
//usage
{{#blog-post title=title}}
<p class="author">by {{author}}</p>
{{body}}
{{/blog-post}}
```

Route (aka Coordinating Controller)

The route object's main purpose is to tell the template which model to display. It does the setup (bootstrapping) and coordination of the other objects in the framework, i.e. the template, the controller, and the model. The route is an object whose responsibility is to translate the URL into application state or more specifically a series of templates, controllers and model objects. If application state changes, the route will transform it back to a serialized form and put that in the address bar. Frequently, the setup of the other objects is done almost automatically by the framework following naming conventions. Given this, it is often the case in Ember that the code needed for a route is simply a function to load the model as shown below.

```
App.ProjectsRoute = Ember.Route.extend({
    model: function () {
        return this.store.find('project');
    },
});
```

Architecturally, the route serves the job of a coordinating controller (a term from its SproutCore roots) which is to bring together the template (view), the controller, and the

model as well as handle more global application wide events. It was named route instead of coordinating controller because everything a route does is driven by the URL.

Another way to think about the job of a coordinating controller is to say that the main responsibilities of a controller are:

1) Load the model
2) Map data from the model to the presentation model (view model) or vice versa
3) Render the view

Routes in Ember do 1 and 3 while controllers do 2. I will go into more detail about that second responsibility in the next section.

Note that the routes model property is just a function and does not actually get run until the controller calls the function to load the model so it might be clearer if it was named "`loadModel`" or "`getModel`".

The route's other job is to receive bubbled events that are application-wide and not specific to a particular template such as signing out of an application, and handle them at a higher level beyond the specific, perhaps nested, template where they might originate.

Controller (aka Mediating Controller)

Ember controllers serve as an intermediary between the view (template) and model. It does the transforming of data; getting the model into the format the view (template) is expecting. If this sounds like a ViewModel or presentation model you are correct. In fact, in this presentation Yehuda Katz, one of the Ember framework authors, said controllers are presentation models. Presentation models and ViewModels are considered synonyms by some developers.

There are some important differences between Ember controllers and the familiar server-side MVC ViewModels. Specifically, an Ember controller is not just an object with properties whose values get copied from the model. Controllers have behavior including action event handlers (functions inside an actions hash/object) that respond to view events. Also the framework inspects the controller's model property and automatically creates a two-way bound property on the controller for each property on the model. So controllers decorate the model. In addition, computed properties that utilize and depend on multiple model properties can be created and they can support data binding as well. In the examples below, save is an action event handler and fullName is a computed property that can be data bound.

```
//action events
<button {{action 'save'}}>Save</button>
App.ProjectsCreateController = Ember.Controller.extend({
    actions: {
        save: function () {
            //save code here
        }
    }
});

//computed property
App.Person = Ember.Object.extend({
    // these will be supplied by `create`
    firstName: null,
    lastName: null,

fullName: function() {
    return this.get('firstName') + ' '
      + this.get('lastName');}.property('firstName', 'lastName')
});

fullName: function() {
    return this.get('firstName') + ' '
      + this.get('lastName');}.property('firstName', 'lastName')
});

var ironMan = App.Person.create({
    firstName: "Tony",
    lastName: "Stark"
});

ironMan.get('fullName') // "Tony Stark"
```

Not all properties in your application need to be saved to the server. Any time you need to store information only for the lifetime of this application run, you should store it on a controller. Examples of non-persistent application state include the current sort or filter applied to a list or table of data, whether a region of the page is expanded, etc.

There are two types of controllers ArrayControllers which represent a collection of models and ObjectControllers which represent a single model.

```
//array controller
App.SongsController = Ember.ArrayController.extend({
    sortProperties: ['name', 'artist'],
    sortAscending: true // false for descending
});


//object controller
App.SongController = Ember.ObjectController.extend({
    soundVolume: 1
});
```

Model

A model in Ember is a class that represents data that needs to be persisted. The data is commonly loaded by a JSON API but Ember is agnostic about how the data is loaded or stored. In our example application we use Ember Data, which is still in beta, to represent the model objects. The model below was created using Ember Data which has the DS root object (DS is an abbreviation for data store). You can specify which attributes a model has by using DS.attr.

```
App.Person = DS.Model.extend({
    birthday: DS.attr('date')
});
```

Attributes are used when turning the JSON payload returned from your server into a record, and when serializing a record to save back to the server after it has been modified. If you don't specify the type of the attribute, it will be whatever was provided by the server. You can make sure that an attribute is always coerced into a particular type by passing a type to attr.

Ember Data provides a nice API for persisting your data.

Ember Data has an in memory data store that is persisted to a backing store when you call `save()` on an object.

```
var post = store.createRecord('post', {
    title: 'Rails is Omakase',
    body: 'Lorem ipsum'
});

post.save(); // => POST to '/posts'
```

If you make changes to an object and then call rollback(), the changes are reverted or undone on the object.

```
post.rollback();
```

You can retrieve persisted records by calling `find()`.

```
var posts = this.store.find('post'); //get a collection of all posts
```

or

```
var aSinglePost = this.store.find('post', 1); // => GET /posts/1
```

Project Model

We'll need a project model to get started.

```
//javascript
App.Project = DS.Model.extend({
    name: DS.attr('string'),
    description: DS.attr('string')
});
```

The model above was created using Ember `Data` and the `DS.attr('string')` is not mandatory but can help when serializing data to persistent storage.

Project Model Fixtures

Ember Data has a built-in feature "fixtures" which allows you to mock data out for your application when you are prototyping.

```
//set the Ember Data adapter to the fixture adapter
App.ApplicationAdapter = DS.FixtureAdapter;

//load the 'mock' project fixture data
App.Project.FIXTURES = [{
    id: '1',
    name: "First Project",
    description: "A description of the first project",
}, {
    id: '2',
    name: "Second Project",
    description: "A description of the second project.",
}];
```

Initially, we will use this feature to load a list of projects. Later in the chapter, we will change Ember Data's adapter to one that supports local storage and remove the fixture data.

Static HTML

Installation

Installation of Ember v1.4.0 used in this example requires you to download the file from emberjs.com and add a script tag to the bottom of the page as shown. Again, the scripts are at the end of the page before the body tag so they don't block the page from loading.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Project Manage Application</title>
    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/style.css">
</head>
<body>

    <script src="js/libs/jquery-1.10.2.js"></script>
    <script src="js/libs/handlebars-1.1.2.js"></script>
    <script src="js/libs/ember-1.4.0.js"></script>
    <script src="js/libs/ember-data.js"></script>
    <script src="js/app.js"></script>
</body>
</html>
```

Ember depends on jQuery for DOM manipulation and Handlebars for templating. In addition, we'll include the Ember Data library to use for persistence when creating the sample.

Templates

The first script tag on the index.html page with an id or data-template-name attribute of "application" or without an id or data-template-name is the "shell" for the application. Other templates on the page in script tags are loaded into the {{outlet}} tag in that application template. You generally don't need to write code to do this. Instead, the framework does it on your behalf using naming conventions.

Below is the markup with the {{outlet}} bolded and the templates italicized so you can visualize how it works. We'll fill in the templates in the next step.

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Project Manage Application</title>
    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/style.css">
</head>
<body>
    <script type="text/x-handlebars">
        <div class="sheet">
            {{outlet}}
        </div>
    </script>

    <script type="text/x-handlebars" data-template-name="hello">
        <p>Hello</p>
    </script>

    <!--scripts omitted for clarity-->
</body>
</html>
```

Templates with HTML

Here is what the project related templates look like with HTML:

```
<!DOCTYPE html>
<html>
<head>
    <meta charset="utf-8">
    <title>Project Manage Application</title>
    <link rel="stylesheet" href="css/normalize.css">
    <link rel="stylesheet" href="css/style.css">
</head>
<body>
    <script type="text/x-handlebars">
        <div class="sheet">
            {{outlet}}
        </div>
    </script>

    <script type="text/x-handlebars" data-template-name="projects">
        <h1>{{#link-to 'projects'}}Projects{{/link-to}}</h1>

        {{#if isAdding}}
        {{outlet}}
        {{else}}
        {{#link-to 'projects.create' }}
        add project
        {{/link-to}}
        {{/if}}

        <ul>
            {{#each project in controller}}
            <li>
                {{render 'project' project}}
            </li>
            {{else}}
            No projects are available.
            {{/each}}
        </ul>

        <hr>
    </script>

    <script type="text/x-handlebars" data-template-name="projects/create">

        <form>
```

```
            {{input type="text" value=newName placeholder="Give the project a
name" }}
            {{input type="text" value=newDescription placeholder="Enter a
description" }}

            <button {{action 'save' }}>Save</button> or
            {{#link-to 'projects'}}cancel {{/link-to}}
        </form>

    </script>

    <script type="text/x-handlebars" data-template-name="project">
        {{#if isEditing }}
        <form>
            {{input type="text" value=project.name placeholder="Give the
project a name" }}
            {{input type="text" value=project.description placeholder="Enter
a description" }}
            <button {{action 'doneEditing' project}}>Save</button> or
            <a href="#" {{action 'cancelEditing' project}}>cancel</a>
        </form>
        {{else}}
        <a href="#" {{action 'edit' project}}>edit</a>
        <a href="#" {{action 'remove' project}}>delete</a>
        {{#if isDeleting}}
        <div class="confirm-box">
            <h4>Are you sure you want to delete this item?</h4>
            <button {{action "confirmDelete" }}> yes </button>
            <button {{action "cancelDelete" }}> no </button>
        </div>
        {{/if}}

        {{project.name}}

        {{/if}}
    </script>

    <script src="js/libs/jquery-1.10.2.js"></script>
    <script src="js/libs/handlebars-1.1.2.js"></script>
    <script src="js/libs/ember-1.4.0.js"></script>
    <script src="js/libs/ember-data.js"></script>
    <script src="js/app.js"></script>
</body>
</html>
```

Notice the `Handlebars template expressions {{ }}` in the templates which will be replaced with dynamic data. We'll get into the details of each template when we discuss its corresponding route and controller.

App Router

Below is the application router for the project features of the example application.

```
App.Router.map(function () {
    this.resource('projects', function () {
        this.route('create');
    });
});
```

We create a root projects route (resource) that will display a list of projects and nest inside it a specific route for creating a project. This gives project create its own url so that a user can share a link to the create a project page. Remember, the first argument "projects" to the `.resource()` function is the route name and the Ember naming convention assumes a route path of "/projects" unless one is passed as the second argument. The third argument is a function where nested routes can be declared for the parent route "projects." The nested route name for creating a project is actually "`projects.create`" based on Ember's naming conventions. The `.resource()` method is used because the `"/projects"` URL describes a noun (project) and the `.route()` method is used because the `"/projects/create"` describes a verb. Below is a screenshot from the Ember Inspector Chrome extension that demonstrates Ember's naming convention including what objects will be created and what they will be named based on the router configuration.

Route Name	Route	Controller	Template	URL
application	ApplicationRoute >$E	ApplicationController >$E	application	
loading	LoadingRoute >$E	LoadingController	loading	#/loading
error	ErrorRoute >$E	ErrorController	error	#/_unused_dummy_
projects	ProjectsRoute >$E	ProjectsController >$E	projects	
projects.loading	ProjectsLoadingRoute >$E	ProjectsLoadingController	projects/loading	#/projects/loading
projects.error	ProjectsErrorRoute >$E	ProjectsErrorController	projects/error	#/projects/_unused_
projects.create	ProjectsCreateRoute >$E	ProjectsCreateController >$E	projects/create	#/projects/create
projects.index	ProjectsIndexRoute >$E	ProjectsIndexController >$E	projects/index	#/projects
index	IndexRoute >$E	IndexController	index	#/

Index Route

At every level of nesting (including the top level or the root '/'), Ember.js automatically provides a route for the / path named index.

For example, if you write a simple router like this:

```
App.Router.map(function () {
    this.route('projects');
});
```

It is the equivalent of:

```
App.Router.map(function () {
    this.route('index', { path: '/' });
    this.route('projects');
});
```

So our example project has an index route and this is the ideal place to redirect the projects route which we want to be the default in our example application.

```
App.IndexRoute = Ember.Route.extend({
    redirect: function () {
        this.transitionTo('projects');
    }
});
```

We call `this.transitionTo` to redirect to the projects route. Note that there is `this.transitionTo` which is called from inside route objects and `this.transitionToRoute` which is available inside controller objects.

Projects Route

The projects route tells the projects template which model to use.

```
App.ProjectsRoute = Ember.Route.extend({
    model: function () {
        return this.store.find('project');
    },
});
```

The model property on the route defines how to load the model data. In this case, Ember Data is available on the route via `this.store` and the `.find('project')` method is called without any additional arguments to find all projects.

Projects Controller

The projects controller is an `ArrayController` because its template will render a list of projects.

```
//projects controller
App.ProjectsController = Ember.ArrayController.extend({
    isAdding: false,
});

//snippet from projects template
<ul>
    {{#each project in controller}}
    <li>
        {{render 'project' project}}
    </li>
    {{else}}
    No projects are available.
    {{/each}}
</ul>
```

We initialize an "`isAdding`" variable to indicate if the template is currently showing the create or add project form. We'll go into more detail on this variable when we go over the ProjectsCreateRoute where we initialize it.

There isn't any other code needed in the Projects controller because the model is loaded in the route and each list item is rendering as part of an individual project template. The projects template uses the #each helper to iterate over the projects and render each by calling the project template using the `{{render}}` helper. The render helper is simply a way of calling a template without needing to change the URL or route. So `{{outlet}}` helpers are used when the URL changes and render can be used in this case because the route does not need to change.

Project Route

This code below is what shows how the project route loads the model.

```
App.ProjectRoute = Ember.Route.extend({
    model: function (params) {
        return this.store.find('projects', params.project_id);
    }
});
```

Ember recognizes this as a common pattern and generates this route code on your behalf. So if you are simply loading a single model you would not need to write this code. In our example application this code is not generated for us or needed because we don't establish a project route because we do in line editing of the project as part of the list and use the model already loaded to populate the list item.

```
{{render 'project' project}}
```

The last argument to render is the current project in the each loop.

Project Controller

The project controller renders an individual list item in the projects list and is called in the projects template using `{{render 'project' project}}` as seen previously.

```
App.ProjectController = Ember.ObjectController.extend({
    isEditing: false,
    isDeleting: false,
    actions: {
        edit: function () {
            this.set('isEditing', true);
        },
        cancelEditing: function () {
            var project = this.get('model');
            project.rollback();
            this.send("doneEditing");
        },
        doneEditing: function () {
            this.set('isEditing', false);
            var project = this.get('model');
            project.save();
        },
        remove: function () {
            this.set('isDeleting', true);
        },
        confirmDelete: function () {
            this.set('isDeleting', false);
            var project = this.get('model');
            project.deleteRecord();
            project.save();
        },
        cancelDelete: function () {
            this.set('isDeleting', false);
        }
    }
});
```

The project template allows three different pieces of functionality.

```
Editing a Project
Deleting a Project
Rendering a Project (read-only)
```

Editing a Project

We need to know when a given project list item is in edit mode. We do this by creating a variable on our controller to hold this current application state:

```
  isEditing: false
```

We also need event handler functions to set this variable to true when someone clicks an edit link for a project.

```
actions: {
        edit: function() {
            this.set('isEditing', true);
        },
        cancelEditing: function(){
            var project = this.get('model');
            project.rollback();
            this.send("doneEditing");
        },
        doneEditing: function() {
            this.set('isEditing', false);
            var project = this.get('model');
            project.save();
        },
}
```

Notice how these events are contained inside an actions hash or object literal: actions{}. This keeps event handlers isolated from other functions so that an event doesn't bubble up and get handled by a function in another controller or route.

We trigger the event by using the `{{action}}` helper on the edit anchor.

```
<a href="#" {{action 'edit' project}}>edit</a>
```

The edit event handler simply sets the "`isEditing`" application state to true so when the controller's (aka presentation model's) application state is changed the template's data

binding will go to work and redraw itself.

```
<script type="text/x-handlebars" data-template-name="project">
  {{#if isEditing }}
       <form>
       //code omitted for clarity
      </form>
  {{else}}
  {{#if isDeleting}}
//code omitted for clarity
    {{/if}}
     {{project.name}}
    {{/if}}
 </script>
```

The doneEditing function is an action called on the save button of the edit form.

```
<form>
{{input type="text" value=project.name placeholder="Give the project a name"
}}
{{input type="text" value=project.description placeholder="Enter a
description" }}
      <button {{action 'doneEditing' project}}>Save</button> or
      <a href="#" {{action 'cancelEditing' project}} >cancel</a>
</form>
```

Inside the `"doneEditing"` function the `"isEditing"` variable is set back to false and Ember Data's `.save()` method persists the model. Note that this is only necessary after we update Ember Data to use the local storage adapter because data binding automatically updates the in memory project model.

The `"cancelEditing"` function is called using an action helper on the button as well (see above). Because two-way data binding is taking place as the user types in the form, canceling the changes by setting the properties of the model back to where they were before editing began requires us to call `Ember Data's .rollback()` function. Lastly, we call `"doneEditing"` to take us back out of editing mode. Notice that for action events to call other events it is necessary to call "doneEditing" using the .send method of the controller.

Deleting a Project

Similar to editing, we need to know when a given project list item is in delete mode. We do this by creating a variable on our controller to hold this current application state:

```
isDeleting: false
```

We also need event handler functions to set this variable to true when someone clicks a delete link for a project.

```
actions: {
        remove:function(){
            this.set('isDeleting', true);
        },
        confirmDelete:function(){
            this.set('isDeleting', false);
            var project = this.get('model');
            project.deleteRecord();
            project.save();
        },
        cancelDelete:function(){
            this.set('isDeleting', false);
        }
}
```

We trigger the event by using the `{{action}}` helper on the delete anchor tag.

```
<a  href="#" {{action 'remove' project}} >delete</a>
```

The remove event handler simply sets the `"isDeleting"` application state to true so when the controller's (aka presentation model's) application state is changed the template's data binding will go to work and redraw itself and show a confirmation message asking the user if they are sure they want to delete.

```
<script type="text/x-handlebars" data-template-name="project">
    {{#if isEditing }}
    <form>
        //code omitted for clarity
    </form>
    {{else}}
    <a href="#" {{action 'remove' project}}>delete</a>

   {{#if isDeleting}}
<div class="confirm-box">
    <h4>Are you sure you want to delete this item?</h4>
    <button {{action "confirmDelete" }}> yes </button>
    <button {{action "cancelDelete" }}> no </button>
</div>
      {{/if}}
      {{project.name}}
  {{/if}}
</script>
```

Depending on the button they click the `"cancelDelete"` or `"confirmDelete"` action event handler will be called. Ember Data deletes by removing from the in memory collection when `.deleteRecord()` is invoked and then persists this change when `save()` is called.

Rendering a Project

Rendering a project is straightforward and done in the template if the project is not currently in one of the other modes `(isEditing, isDeleting).`

```
<script type="text/x-handlebars" data-template-name="project">
  {{#if isEditing }}
       <form>
       //code omitted for clarity
      </form>
{{else}}
{{#if isDeleting}}
//code omitted for clarity
{{/if}}

    {{project.name}}

    {{/if}}
  </script>
```

Projects Create Route

When the add project link is clicked on the projects (list) template, the `{{link-to}}` block helper sends the user to the `"projects.create"` route.

```
{{#if isAdding}}
{{outlet}}
{{else}}
{{#link-to 'projects.create' }}
add project
{{/link-to}}
    {{/if}}

        App.ProjectsCreateRoute = Ember.Route.extend({
            activate: function(){
                this.controllerFor('projects').set('isAdding', true);
            },
            deactivate: function(){
                this.controllerFor('projects').set('isAdding', false);
            }
        });
```

When the projects create route is created an activate function hook is called on the route. Inside the activate function we set the `"isAdding"` application state to true on the projects controller so that the project list view knows to show the form for adding a project. Remember we initialized the "isAdding" variable to false in the `ProjectsController` previously.

```
App.ProjectsController = Ember.ArrayController.extend({
    isAdding: false,
});
```

When a user leaves the "projects/create" route by clicking cancel the deactivate function hook is called on the controller which sets the adding mode back to false.

Once `"isAdding"` is set to true when the user navigates to `"projects/create"` the

associated template is placed into the `{{outlet}}` since `projects.create` is a child route of projects (notice how the create route is nested under the parent "projects" in the router definition below).

```
{{#if isAdding}}
{{outlet}}
{{else}}
{{#link-to 'projects.create' }}
add project
{{/link-to}}
    {{/if}}


App.Router.map(function () {
    this.resource('projects', function () {
        this.route('create');
    });
});
```

Projects Create Controller

When users navigate to the `"projects/create"` route they will see the following form:

```
<script type="text/x-handlebars" data-template-name="projects/create">

    <form>
        {{input type="text" value=newName placeholder="Give the project a
name" }}
        {{input type="text" value=newDescription placeholder="Enter a
description" }}

        <button {{action 'save' }}>Save</button> or {{#link-to
'projects'}}cancel {{/link-to}}
    </form>

</script>
```

If they click the save button the `ProjectsCreateController` will handle the action with the 'save' event handler function.

```
App.ProjectsCreateController = Ember.Controller.extend({
    actions: {
        save: function () {
            var newProject = this.store.createRecord('project', { name:
this.get('newName'), description: this.get('newDescription') });
newProject.save();

            this.set('newName', '');
            this.set('newDescription', '');
            this.transitionToRoute('projects');
        }
    }
});
```

The save() function uses Ember Data to create a new record in memory: createRecord() by passing in the newName and newDescription properties which get automatically created on the controller. As we've seen previously, calling "save" on the model object will persist the data.

Local Storage Data Adapter

If we download an Ember Data local storage adapter such as this one from Ryan Florence, and then include it in the scripts section at the bottom of the index.html page.

```
<script src="js/libs/jquery-1.10.2.js"></script>
<script src="js/libs/handlebars-1.1.2.js"></script>
<script src="js/libs/ember-1.4.0.js"></script>
<script src="js/libs/ember-data.js"></script>
<script src="js/libs/ember-data-localstorage-adapter.js"></script>
<script src="js/app.js"></script>
</body>
```

And also update the application's adapter and remove the fixture data as shown below our application will now be able to persist data across browser sessions.

```
App.ApplicationAdapter = DS.LSAdapter.extend({
    namespace: 'projectmanage-funnyant'
});


App.Project.FIXTURES = [{
    id: '1',
    name: "First Project",
    description: "A description of the first project",
}, {
    id: '2',
    name: "Second Project",
    description: "A description of the second project.",
}];
```

When we first reload the page our projects will be missing but as we add new ones they should stick around because they are being persisted to the web browser's local storage.

Challenges

As I did previously with the other frameworks, in this section I'm going to move faster and discuss the challenges I ran into as the example application became complex.

Learning Curve

Although the Ember team has done a lot of work to make the "getting started" demos impressive with easy two-way data binding like AngularJS, you still need to learn a lot about the framework in order to use it to build an application. More specifically, you need to understand the architecture as a whole including: the naming conventions, what is done in the route objects, how nested routes and outlets work, the difference between a resource and a route inside the router, etc. It becomes a steep learning curve to be productive but after you get over it you are more productive than with any of the other frameworks.

Accessing Parent Controllers and Routes

Nested routes are a great feature but as hierarchy is created in your user interface there is still a need to communicate between levels in the hierarchy. For example, you'll frequently find yourself in a child controller or route and need access to its parent's controller or model.

If you are in a controller this is done through the needs property as follows:

```
needs: 'projectsdetail'
```

After telling the framework that access is needed, you can then access the controller from the controllers property.

```
App.ProjectsdetailListsCreateController = Ember.Controller.extend({
    needs: 'projectsdetail',
```

```
    actions: {
        save: function () {
            var project = this.get('controllers.projectsdetail.model');

            //code omitted for clarity
        }
    }
});
```

If you are in a route and need access to the controller you use the controllerFor method (note that this is not available when you are inside controllers). We have seen this in all our "create" routes. Below is an example, but note that you have access to any controllers when inside a route object, not just your own controller as shown here.

```
App.ProjectsCreateRoute = Ember.Route.extend({
    activate: function () {
        this.controllerFor('projects').set('isAdding', true);
    },
    deactivate: function () {
        this.controllerFor('projects').set('isAdding', false);
    }
});
```

Naming Conventions

The Ember naming conventions are obvious but do require some initial learning to understand the nuances, particularly when default index routes are introduced or dot syntax is used. For example, I named a resource `"projects.detail"` and decided to change it to `"projectsdetail"` (without the period) and needed to change my controller and route names from "`ProjectsDetail`" to `"Projectsdetail."`

Hierarchy

As with the other frameworks, I experienced a lot of challenges as I started working with hierarchical data (projects with lists and each list with todos). To be fair, most of my challenges were in learning Ember Data and not with the Ember framework itself. Because

Ember Data is aware of one-to-many relationships it was much easier to get the example working than my experiences with the other frameworks.

As mentioned earlier, I used an open source local storage adapter for Ember Data. I lost quite a bit of time before realizing I needed to not only override the storage adapter but also the serializer for Ember Data. Once I set the serializer object, child objects persisted properly.

```
App.ApplicationSerializer = DS.LSSerializer.extend();
App.ApplicationAdapter = DS.LSAdapter.extend({
    namespace: 'projectmanage-funnyant'
});
```

When creating a new list I had to be careful to not only add the list to its parent project "lists" collection but to also set the parent project property on the list.

```
var project = this.get('controllers.projectsdetail.model');
var newList = this.store.createRecord('list', { name:
this.get('newListName'), description: this.get('newListDescription'),
project: project });
newList.save();

var lists = project.get('lists');
lists.pushObject(newList);
project.save();
```

Lastly, when I deleted a list, I found you have to manage the parent relationship manually and carefully. In my experience, other server-side object relational mapping (ORM) tools have done more of this work on the developer's behalf.

```
var list = this.get('model');
var project = this.get("controllers.projectsdetail.model");
project.get('lists').removeObject(list);
project.save();
list.deleteRecord();
list.save();
```

Debugging

Ember has great support for promises and asynchronous programming. Unfortunately, it is fairly common to end up in an asynchronous callback function where the call stack for the action that triggered the exception is not available. Given that there is so much auto-wiring of objects based on the framework naming conventions this can make some bugs difficult to track down because you're not sure where they originated. I think it is important to note that the actual error messages that are thrown when debugging are the most helpful of any of the frameworks. In fact, the error messages were frequently so good that I didn't have to set a break point or debug further because I immediately understood and could fix the error being reported.

Wrapping plugins and widgets

Wrapping existing plugins and widgets such as a jQuery UI Datepicker inside an Ember component was challenging in particular getting the control to work with the existing two-way data binding in the framework. The experience was very similar to the one I had using directives in AngularJS to do the same task but slightly better due to the more straight-forward API provided by Ember.

Conclusion

Despite the challenges, I had the most productive and rapid application development experience using Ember. However, I did spend more time upfront understanding Ember's architecture than with the other frameworks. Also Ember having a data project (Ember Data), particularly one that acknowledges you'll have hierarchical data in your application, gave it a perhaps unfair but noticeable advantage over AngularJS which leaves data persistence and models mostly out of scope.

CHAPTER 9

CHOOSING A FRAMEWORK

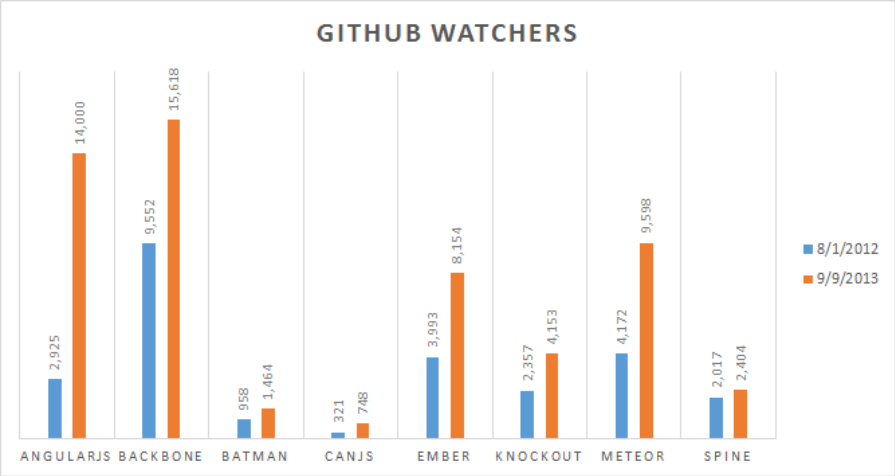
Introduction

The frameworks I'll be considering in this discussion are the ones with the most traction at present: AngularJS, Backbone, Ember, and Knockout. Batman, CanJS, Meteor, and Spine are also mentioned but not covered in depth.

Each project is examined from several different perspectives including community, leadership, maturity, size, dependencies, interoperability, inspiration, philosophy, and features.

Community

A good indicator of the health of any open source project is its community. The table below shows the number of GitHub watchers for each of the projects.



You wouldn't want to make your framework decision on this data alone but it certainly gives you a sense of which frameworks are:

Most established

- Backbone.js
- AngularJS

Experiencing the most growth (in the last year)

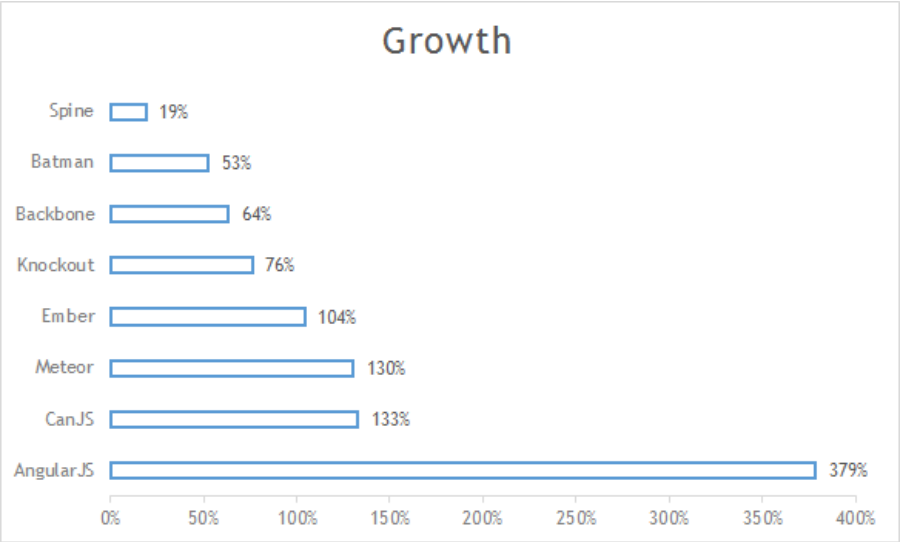
- AngularJS
- Meteor
- Ember
- Knockout

Showing a small following but growing rapidly

- CanJS

Growth

In particular it is worth noting the incredible growth of AngularJS (379%) in the past 13 months and taking this into consideration when making your decision. The chart below compares the growth of GitHub watchers (over that 13-month period) to provide an idea of how fast the community has been growing for each project. Meteor (130%), Ember (104%), Knockout (76%), and Backbone (64%) also have had amazing growth considering their previously sizable communities.



Leadership

Understanding the people, their backgrounds, and the problems they were trying to solve when they created a framework helps you appreciate their design decisions and motivations. For example, David Heinemeier Hansson, creator of the popular Ruby on Rails web development framework, was working as a contract developer for 37signals design projects and had only 10 hours a week to work on the framework. Ruby on Rails was actually extracted from some of his initial contract work with 37signals. This background helps you

understand that the framework had to be extremely productive for the developer which means a lot of conventions (decisions already made) and scaffolding (generated code). Below, I'll introduce you to the creators of the JavaScript MVC frameworks so you might develop a similar appreciation for their work.

Backbone

Jeremy Ashkenas and DocumentCloud

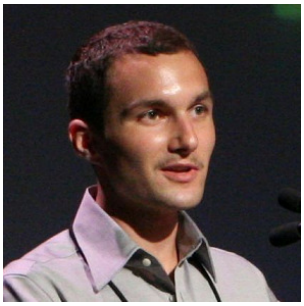


Image Credit: The Canadian University Software Engineering Conference

Jeremy Ashkenas is the creator of the CoffeeScript programming language, Backbone.js JavaScript framework, and Underscore.js a JavaScript utility library. According to Wikipedia, he is currently working in Interactive News at the NYTimes /DocumentCloud.

AngularJS

AngularJS was originally developed at Google in 2009 by Miško Hevery and Adam Abrons as the software behind an online JSON storage service. Abrons left the project, but Hevery, who works at Google, continues to develop and maintain the library with fellow Google employees Igor Minár and Vojta Jína.

Igor Minár and Miško Hevery, Image Credit: Devoxx 2012

Knockout

Steve Sanderson is the original creator of Knockout. Steve Sanderson is currently working as a developer for Microsoft in the team that brings you the ASP.NET technology stack, IIS, and other web things. Previously, he developed .NET software as a contractor/consultant for clients in Bristol and beyond, plus wrote some books for Apress, such as Pro ASP.NET MVC Framework.

Image credit: Steve Sanderson's Blog

Ember

The two most well-known and public members of the Ember core team are Yehuda Katz and Tom Dale.

Yehuda Katz is a member of the Ember.js, Ruby on Rails and jQuery Core teams; He spends his daytime hours at the startup he founded, Tilde Inc.. Yehuda is co-author of the best-selling jQuery in Action and Rails 3 in Action books.

Tom Dale was previously on the SproutCore team. He's a former Apple software engineer who gained expert front-end JavaScript skills while working on the MobileMe and iCloud applications.

Image Credit: Ember.js website

Meteor

The Meteor development group just raised $11.2 million so they can do this fulltime and they have a team of 12 developers with impressive resumes. The team has ambitious goals that stretch beyond the scope of most JavaScript MVC frameworks which focus on organizing client-side code and state. Meteor is a full-stack framework including architecture on the web server and the database.

CanJS

CanJS was created roughly 2 years ago by Justin Meyer and his team at Bitovi, a web application consulting company. CanJS was extracted from the company's original framework JavaScriptMVC which has existed for 6 years at the time of this writing. Bitovi's core business is building applications with the CanJS framework.

Maturity

Considering how mature each framework is helps you understand how big of a risk you are taking when using these newer technologies in your project. New and unproven frameworks can have problems with documentation, scalability, stability (API changes), and support (finding developers to maintain the code who know the framework) that could cause an otherwise good decision to backfire. Some things to consider include: How many real-world production apps are using these frameworks and how many users do these apps have? How good is the documentation and how many examples/tutorials are available? Are the examples up to date? How stable is the API? Do other developers know or are they getting to know this technology?

- Backbone (most mature)
    - apps in production for 3 years now including GroupOn, FourSquare, USAToday, DocumentCloud, etc...
    - good documentation
    - good examples but many now outdated
    - API very stable
    - lots of watchers on GitHub

- AngularJS (mature)
    - in production now at Google but does not have as long a track record as other

projects
    - fair documentation, getting better
    - lots of examples
    - lots of watchers on GitHub

- Knockout (mature)
    - in production for 2 years now
    - great documentation including JSFiddle like examples
    - API stable
    - lots of examples
    - lots of watchers on GitHub

- Ember.js
    - first production release 1.0 on August 30, 2013 after 2 years of development
    - documentation improving but hard to find solutions to problems
    - API had intentionally not been stable until 1.0 release
    - good number of examples some outdated due to API changes prior to 1.0
    - lots of watchers on GitHub

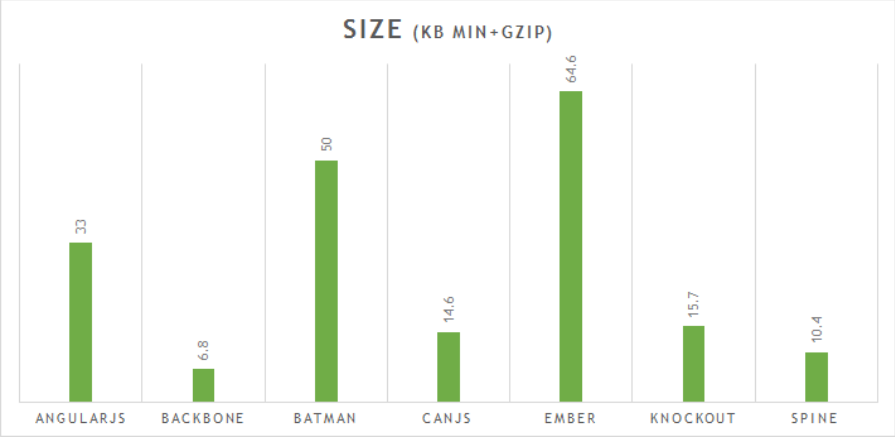
- Meteor
    - still early in development used mostly in example apps
    - documentation good but a work in progress
    - API still evolving
    - some examples
    - lots of watchers on GitHub

- CanJS
    - Roughly 2 years old but extracted from framework that is 6 years old
    - Applications in production for Walmart, Cars.com, Apple (store.apple.com),

Sams Club mobile app, Wanderfly

Size

It's important to understand how big a download each of these frameworks is and what you are getting for that extra weight in your application. The size affects performance but also gives you an indication of how ambitious a framework is and how long it might take you to learn this new technology as well as hint at how many ways it's trying to help you build your application (i.e. how many features does it support and how robust are they). The more ambitious and feature rich a framework is the harder it will typically be to integrate it with others particularly on the same page of an app. Lighter frameworks are more like a library and a smaller commitment to include it in your project.



Some of these projects such as Backbone and Spine pride themselves on how small they are and think of themselves more as a library than as a framework. Often these smaller frameworks leave room for "choosing your own" library to use for features such as templates and routing. We'll explore this in more detail when we talk about the features of each.

Other projects, such as Ember and AngularJS are more ambitious and are comfortable being called a framework. They often have more built-in features and depend less on external libraries.

Below is a list showing which projects are considered more of a library versus a framework.

Libraries	Frameworks
Backbone	Ember
Knockout	AngularJS
Spine	Batman
CanJS	Meteor

Dependencies

What other libraries are required to build a real-world application with these projects? The chart below takes a look at what the average number of dependencies each library requires for the developer to be productive and looks at size including these dependencies.

These numbers were gathered by downloading libraries from cdnjs. In practice, most projects will use jQuery with these frameworks to do DOM manipulation in a web application because they need animation and AJAX support as well. In a mobile application it's not uncommon for projects to use Zepto.js which is a much lighter library for handling DOM manipulation but doesn't support Internet Explorer which is not a common requirement for mobile applications. AngularJS already has trimmed down version of jQuery jQLite included but jQuery can override it if used in your project. The AngularJS team encourages developers to not add the full jQuery library unless needed. To help you make the right choice, the table above shows a mobile column which assumes Zepto.js and a web application column which shows jQuery.

Web App	
	Sum of Size (kb min+gzip)
AngularJS	**33**
angular.js	33
Backbone	**44.8**
underscore.js	5.4
backbone.js	6.8
jquery.js	32.6
Batman	**50**
batman.js	50
CanJS	**47.2**
canjs.js	14.6
jquery.js	32.6
Ember	**111.4**
ember.js	64.6
handlebars.js	14.2
jquery.js	32.6
Knockout	**58.7**
sammy.js	6.6
knockout.js	15.7
knockout.mapping.js	3.8
jquery.js	32.6
Spine	**48.4**
underscore.js	5.4
spine.js	10.4
jquery.js	32.6

Mobile App	
	Sum of Size (kb min+gzip)
AngularJS	**33**
angular.js	33
Backbone	**22.4**
underscore.js	5.4
zepto.js	10.2
backbone.js	6.8
Batman	**50**
batman.js	50
CanJS	**24.8**
zepto.js	10.2
canjs.js	14.6
Ember	**78.8**
ember.js	64.6
handlebars.js	14.2
Knockout	**36.3**
sammy.js	6.6
knockout.js	15.7
knockout.mapping.js	3.8
zepto.js	10.2
Spine	**26**
underscore.js	5.4
zepto.js	10.2
spine.js	10.4

Interoperability

This section discusses whether each framework is designed to control the whole page or if it can be used as a small part of an existing page as you slowly work this new technology into an existing project. The earlier library or framework discussion for the most part determines how interoperable each of these projects is...so the libraries tend to be easier to integrate into existing projects while the frameworks do more for you but don't play as well with others.

AngularJS

Works well with other libraries but developers are encouraged to see if they can do without jQuery and jQueryUI. In fact Angular has a subset of jQuery called jqLite. The rationale for following this practice is ease of unit testing as many dependent libraries and plugins weren't designed with unit testing in mind and are subsequently more difficult to mock away in unit tests. In practice most developers end up using jQuery for something and including it.

Backbone

Because of its small footprint and un-opinionated architecture it's easy to include with numerous popular client-side libraries and server side technologies.

Ember

Intended to control your whole page at run-time so not suited for use on only part of a page.

Knockout

Can be used as a small part of larger projects and doesn't need to control the whole page.

CanJS

CanJS plays extremely well with third-party UI library controls including jQuery UI and DOJO allowing the controls to be properly disposed avoiding memory leaks that can plague single-page applications.

Inspiration

A favorite question of journalists interviewing musicians is: "What artists did you listen to growing up or who inspired you?" This often leads to insights or gives hints to their readers of what sound they can expect from that musician. Most of the ideas in these frameworks are not new to software development but come from technologies the creators had worked on in the past and enjoyed. This section summarizes what information I could find from interviews with the creators of these frameworks about their inspiration.

AngularJS

Declarative programming languages such as HTML and the rich internet application technologies (RIAs) such as Flex from Adobe and Windows Presentation Foundation (WPF) and Silverlight from Microsoft were technologies the creators of AngularJS were heavily influenced by in their work. These declarative technologies don't have a "main" method and just express what needs to happen but don't specify the implementation. Two-way data binding in views to model objects is a canonical example of this declarative programming style in action. Also dependency injection and inversion-of-control (IOC) containers in particular Juice which is used heavily in server-side Java code by Google engineers is a stated inspiration for the creators as they value unit testing and need a framework that is designed to allow you to inject your dependencies so tests can be isolated from other application layers and run fast.

Ember

Tom Dale did a great job describing Ember's influence on Quora:

"With Ember.js, we've spent a lot of time borrowing liberally from concepts introduced by native application frameworks like Cocoa. When we felt those concepts were more hindrance

than help--or didn't fit within the unique constraints of the web--we turned to other popular open source projects like Ruby on Rails and Backbone.js for inspiration. Ember.js, therefore, is a synthesis of the powerful tools of our native forebears with the lightweight sensibilities of the modern web."
–Tom Dale on Quora

In addition, it's important to understand that Ember.js is an evolution of the SproutCore JavaScript library and became Ember at the point when SproutCore stopped becoming more Cocoa like and was more heavily influenced by jQuery.

Knockout

This Hanselminutes podcast has some great background on Steve Sanderson's inspiration. In summary, the Model View View Model (MVVM) Design Pattern and declarative technologies such as Microsoft's WPF (Windows Presentation Foundation) and Silverlight were the biggest inspirations. You may have noticed that the declarative two-way data binding that is the best feature of Knockout is very similar to AngularJS because they had some of the same influences.

CanJS

According to Justin Meyer's Rails was a big influence in particular with the naming and API. The evolution of the framework particularly the recent features added in 2.0 have been influenced by other JavaScript MVC Frameworks. More specifically, Angular's two-way binding and directives (custom elements in CanJS).

Philosophy

Newspapers generally strive to be unbiased in their reporting of the news. The one exception to this is the editorial section where opinions are encouraged and writers often take a strong position on an issue. At times both types of writing are not strictly unbiased reporting or strong opinion but somewhere in the middle of this continuum. Technology frameworks have a similar division tending to be more strongly opinionated or not as opinionated. For example, Ruby on Rails values convention over configuration and makes lots of decisions on behalf of the developer such as file structure and data access. Consequently, it is considered to be pretty strongly opinionated. Other server-side frameworks such as Sinatra are more light-weight and not opinionated about file structure and data access. Consequently, they are viewed as not as opinionated. Just as these server-side frameworks have philosophies the client-side JavaScript MVC frameworks we've been discussing can be examined in the same light on this continuum of opinionated to not opinionated. Let's look at each of the projects and discuss their philosophy.

AngularJS: opinionated

Pretty opinionated in particular their emphasis on testability and dependency injection. Also, the idea that declarative programming such as HTML is awesome are pervasive in the framework.

Ember: Extremely opinionated

Strives for developers to only make decisions about what is different for their application and take care of the rest through convention and scaffolding. This philosophy is similar to the Ruby on Rails and jQuery and is best expressed by the emberjs.com website:

Don't waste time making trivial choices. Ember.js incorporates common idioms so you can

focus on what makes your app special, not reinventing the wheel.

Ember standardizes files and URL structures but does allow you to override these things if needed. Expect to get a lot more code generated for you and a lot more conventional ways of doing things such as file structure. Consequently, you'll need to make less mundane decisions because the framework has already chosen a reasonable default and you can get down to the business of building the unique parts of your application.

Knockout : unopinionated

Leaves routing and data storage to the developer's choice. Doesn't dictate file or URL structure. Even allows declarative DOM-based templates to be replaced with string-based templates.

Features

Think of these various Javascript MVC Frameworks as a set of common features to help developers build single-page apps. The way each framework implements these features or chooses not to implement these features and instead makes pluggable another library to complete the framework provides critical insight.

So what are the main features of a Javascript MVC Framework?

- Two-way Binding between HTML and a client-side JavaScript object model
- Templates/Views
- Models
- Routing (keeping the back button working and search engines happy)
- Data Storage (local or through a web server to a database)

In addition, some of the frameworks provide common language services such as generic pub/sub event model and support for object-oriented inheritance.

Data binding

This is the most touted feature of these frameworks. You change the data in an HTML input and the JavaScript object bound to that input is immediately updated as well as any other user interface elements that are bound to that same property. In many of the frameworks, it goes the other way as well, if you change the JavaScript object the html will automatically refresh. Its two-way data binding on the web as you've always experienced in rich client application frameworks such as Flex, Windows Forms or Windows Presentation Foundation (WPF). Below is a table showing which frameworks support data binding.

Framework	Data-binding
AngularJS	two-way
Backbone	none
Batman	two-way
CanJS	two-way
Ember	two-way
Knockout	two-way
Meteor	two-way
Spine	none

Some people might argue Backbone and Spine have some support for data binding but there is enough work left to the developer that I feel it's safe to say its not a feature of these libraries.

Templates/Views

The client-side JavaScript model data needs to get interspersed with the HTML and these frameworks take one of two approaches to solving this problem.

String-based templates, of which the most popular is currently handlebars.js, take string or text templates and replace the dynamic parts with data from your models. One of the frequently cited but debatable advantages to string-based templates is performance. The cons seem to be difficulty debugging flow of control type logic.

DOM-based templates embrace the declarative nature of mark-up and create an html on steroids experience where html is annotated with additional attributes to describe the binding and events needed. These libraries require substantially less code but do substantially more magic on the developer's behalf.

Framework	View Templates	Commonly Used
AngularJS	Declarative DOM-based (mandatory)	
Backbone	String-based (choose your own)	underscore.js, handlebars.js
Batman	Declarative DOM-based (mandatory)	
CanJS	String-based (mandatory)	EJS4 and mustache.js
Ember	String-based (mandatory)	handlebars.js
Knockout	Declarative DOM-based (optional)	can do string based
Meteor	String-based (mandatory)	
Spine	String-based (choose your own)	underscore.js, handlebars.js

Models

Some frameworks (Backbone, Spine) are more focused on the model and ask the developer to extend their JavaScript model classes from a base model type and access all properties via .get() and .set() methods so change tracking can happen and events can be triggered when the model changes. Knockout has the developer apply observable wrappers around your plain old JavaScript objects and then has you access properties via object.propertyName() syntax (notice the parentheses).

Other libraries (AngularJS) do dirty checking on all bound DOM elements on the page since there are no standard get and set accessors. Which leads to the performance concern that these libraries will have problems on larger pages. Not only do these libraries require less code to refresh the templates, they also don't require you to use specific get or set accessors to change data so your models can just be plain old JavaScript objects. This results in much higher developer productivity, particularly when first getting started with these frameworks.

Data Storage

These frameworks store data on the server by

- automatically synchronizing with RESTful services
- asking the developer to implement do-it-yourself AJAX calls to web services returning JSON
- allowing either of the above approaches

REST

Some of the frameworks by default assume you have an extremely clean RESTful JSON service on the server and that you (at least by default) are pretty chatty with that service updating data asynchronously in the background while the user interface continues to respond to the user. Internally, these frameworks use jQuery or Zepto to send the appropriate AJAX request to the server. Just as the user interface's HTML DOM elements listen for changes to the JavaScript object model for the application, the sync implementation gets notified of changes to properties on the model and sends updates to the RESTful service to keep the model in "sync" on the server.

Connected or Disconnected

Backbone by default sends the requests before the data is saved client-side so the server and client stay in sync more easily. Spine, a very similar framework to Backbone, takes a different approach by saving records client-side before sending the request asynchronously to the server which provides a more responsive user interface and allows for disconnected scenarios frequently found in mobile applications. If your project requires disconnected scenarios, be sure to understand how well the framework you're using supports this feature.

Do-it-yourself (DIY)

These frameworks ask the developer to use $.ajax (jQuery) to make web services calls or add

another complimentary open-source library to handle data storage needs.

Data Store Specific

More elaborate frameworks such as Meteor have a much more complete data storage story but mandate a MongoDB database on the server. They do this in an effort to provide an incredibly scalable solution by default and support a JavaScript from top to bottom development experience.

See the table below for a summary of how each framework approaches data storage.

Framework	Data Storage	Comments
AngularJS	Manual AJAX or Restful Client	overridable to accommodate different server dialects
Backbone	Model sync method (overridable)	assumes Restful service
Batman	Built-in (mandatory)	
CanJS	Built-in (optional)	
Ember	Built-in (overridable)	
Knockout	Manual AJAX or choose your own	e.g., knockout.mapping or just $.ajax
Meteor	Built-in (mandatory)	requires MongoDB
Spine	Model sync method (overridable)	

Routing

Maps URL routes to JavaScript functions to allow for back button support in browsers. One major downside of single-page applications is that because the page doesn't refresh no entries get added to the browser's history so the back button frequently doesn't take the user back to the previous page state without the developer doing some extra work during those major transitions of state and implementing a state tracking mechanism through a hash appended to the URL or using the push state and pop state in modern browsers. In summary, most projects either provide very basic rudimentary but useful functionality in this area. Knockout simply allows you to plug in another third-party open source library. An exception with routing seems to be Ember which at one point during their project took community feedback and went back to the drawing board to get this right before stabilizing on version 1.0.0. CanJS also has a more elaborate router that does more than map functions to routes and can maintain more elaborate "states" in an application.

Apples to Apples

After looking at the projects by features it became clear to me that I wasn't really comparing "apples to apples." A more fair comparison might be to compare full frameworks like AngularJS and EmberJS with MV* Stacks that include great but less broad libraries like Backbone and KnockoutJS. To be more specific, the following comparisons would be more "apples to apples":

AngularJS
EmberJS
Backbone with Marionette
KnockoutJS with DurandalJS, and HistoryJS or SammyJS
CanJS

Summary

There is a lot to consider when choosing a JavaScript MVC Framework for a project but hopefully I've given you a jump start and lots of context to help wrap your head around these great new technologies.

CHAPTER 10

PROS AND CONS

Introduction

When making a big decision a popular tactic to help is to write down a pro and con list. This chapter is a pro and con list for each framework along with a more detailed description of what each pro or con means. Reviewing this chapter should be a reminder that no framework is perfect despite what you read and each has its strengths as well as weaknesses.

AngularJS Pros

- Testing
  - AngularJS is known for encouraging both unit and functional testing. More specifically, it encourages unit testing by having built-in dependency injection so it's easy to mock away dependencies, even things that are commonly difficult to mock away including HTTP requests and interactions with the DOM are easy to isolate in the AngularJS architecture. Tom Dale, one of the creators of Ember, was asked one thing he admired about AngularJS and he brought up unit testing. AngularJS's functional/integration testing before Angular 1.2 was called "scenario runner" but newer versions of the framework use the Protractor project. Despite this change, functional/integration testing has always been a priority as well.
- Productivity
  - Data binding is the most prominent example of the productivity increase you get with AngularJS. Review the data binding section of the "How to Learn Frameworks Quickly" chapter to see the dramatic reduction in code that happens with data binding.
- Vision
  - AngularJS has a clear path to integrate future web standards in particular the new web components standards. Directives fill the same needs as the web components standards and the Angular team has been working with the Chrome team to help and influence the standards themselves.
- Leaves jQuery behind
  - AngularJS is not just a wrapper around jQuery code; it is a true evolution in web development. The id and class properties commonly used to store application state are almost entirely unnecessary when developing with AngularJS. It is common to bring in the full jQuery library (jqLite is frequently not enough) to help with DOM manipulation but the manipulation can realistically be isolated to directives.
- It's not an abstraction on top of HTML, you can just write HTML
  - Look at the templates in an AngularJS application and you feel like you're just reading

HTML. This is not the case with other frameworks.

- ☐ All in one package, no dependencies
  - o With the exception of the angular-ui router everything you need to build a real-world application is in one library.
- ☐ Has Biggest Community, Clear Leader in Race
  - o AngularJS has such a lead in the community and popularity race that many are already declaring it the victor. It is probably too soon to declare AngularJS a clear winner but its lead in the race is definitely a big positive.
- ☐ Can find answers to questions
  - o Stackoverflow.com has 33,415 AngularJS questions. In comparison Backbone has 14,524 and Ember has 9,183. There are many possible explanations for this disparity. AngularJS is hard some might say or Ember has not been out as long. These are valid arguments but there is no question that when you are stuck with AngularJS you will be able to find an answer easily.
- ☐ No getters and setters, POJOs
  - o Angular feels closer to using raw JavaScript because you use plain old JavaScript objects (POJOs). This does have performance consequences but makes life significantly easier for developers. It is common to forget to call .get() or .set() when accessing or setting properties with other frameworks.
- ☐ Data binding Syntax is Simple
  - o There is a reason Handlebars.js won the template library popularity contest, it has a very simple syntax {{dynamic-stuff}}. Other frameworks have helpers but Angular just asks you to write HTML and intersperse dynamic content with the handlebars or use a directive.
- ☐ Easiest to learn
  - o Many people say AngularJS is hard to learn but for me it was the easiest to learn of all the frameworks. I do think I'm probably in a minority and many would say Backbone is easier but I had trouble understanding Backbone at first because I wanted the library to be more opinionated and to do more.

- Most similar to server-side MVC conceptually
  - Most people come to these frameworks from a server-side background using an MVC Framework like Rails, Cake PHP or ASP.NET MVC and look for corresponding concepts and are frequently confused. AngularJS most clearly maps to those familiar concepts although it is still not an exact match.
- Backed by Google
  - Being and open source project backed by Google means people aren't working on this project in their spare time, and among other things has a more rigorous testing and release process. There are cons to the Google backing and support as well, which will be mentioned in the next section.

AngularJS Cons

- Router
  - The basic router functionality that comes with AngularJS 1.x is known as ng-route. It has several limitations that were acknowledged in the Angular team's design document for Angular 2.x and ng-route was actually removed from the framework in version 1.2 and higher. In the Angular team's words: the initial Angular router was designed to handle just a few simple cases. As Angular grew, we've slowly added more features. However, the underlying design is ill-suited to be extended much further.
    - It's doesn't make it easy for developers to write apps that ignore URLs and break the back button with Angular. Angular should make it easier for developers to create apps with deep linking.
    - There is no support for nested or sibling views.
    - There is no support for state-based routing.
  - Most people use the AngularUI project's router in their AngularJS applications which solves most of these issues and can lessen the impact of these problems.
- Minification can break application if not done properly
  - Since Angular infers the controller's dependencies from the names of arguments to the controller's constructor function, if you were to minify the JavaScript code for a controller, all of its function arguments would be renamed by the minifier, and the dependency injector would not be able to identify services correctly. We can overcome this problem by annotating the function with the names of the dependencies, provided as strings, which will not get minified. However, the syntax for doing this is difficult to read and makes code harder to maintain and comprehend. A "pre-minifier" like ngmin can be used to modify Angular code to prevent errors that may arise from minification while still keeping the clean syntax and is the best practice solution to this problem.
- Lack of Model
  - Although it is liberating for your models to just be plain old JavaScript objects it can

also be confusing because the familiar object-oriented concepts like classes and inheritance are missing from JavaScript. In addition, it can be helpful to have a base class for your models which establishes where validation and change tracking should happen. Validation and change tracking are not missing from AngularJS but can be difficult to find and wrap your head around. Lastly, AngularJS has the concepts of services, factories and providers and it is confusing as to whether these should be used to help represent the business model.

- Integrating third-party controls
  - Using jQuery UI plugins or other third-party controls can be challenging and it is often easier to just roll your own directive (modal dialogs) but in some cases, such as using a calendar control, rolling your own is too big an effort so you are left trying to integrate an existing one. Integrating plugins is challenging because they are not integrated with the AngularJS digest cycle used to achieve automatic two-way data binding so the developer is left to wrap them in a directive and notify AngularJS when the values change. It's not that hard but can be confusing at first. Using plugins from the AngularUI project can make this easier.
- Your jQuery powers are taken
  - AngularJS encourages DOM manipulation to be isolated in directives so they can easily be tested. This is great but often results in developers initially feeling that their jQuery powers to find and change anything on the page have been taken and things that were previously easy such as setting focus on a control can become much more complicated.
- Search-Engine Optimization (SEO) can be hurt
  - Most search engine crawlers have little to no support for running JavaScript on a page before indexing it. Since these frameworks all put the page together in the browser using JavaScript, search engines crawling pages using these frameworks would see a mostly empty page. There are numerous solutions commercial and open-source that usually involve detecting on the server-side that the user is a bot or crawler (using the User-Agent string sent in the HTTP header of the request) and running a light weight

browser instance on the server to render the page (including running the JavaScript) and then returning the resulting HTML. This "con" is not specific to AngularJS but applicable to all the frameworks and something you should be aware of. I would go as far as to say if I had a public-facing brochure-ware site I would not recommend using one of these frameworks. This article does a great job getting into the details of why SEO can be troublesome with a client-side framework.

- Two-way binding not as useful as advertised
  - In practice, I've found one-way data binding which updates the view when changes are made to the model to be extremely useful. Data binding from the view to the model real-world use cases are less common (for example, you want a live preview of how text will look as the user types text in an editor).
- Performance of watchers ($watch)
  - It is pretty easy to add a watch to an object or array of objects and have it cause performance problems in your application.
- Persistence
  - Beyond AngularJS's humble $http, many developers have desired a higher level abstraction for working with data from servers and local persistent data in the browser.
- Data binding performance
  - AngularJS uses dirty checking to detect if changes occurred to an object bound to the UI or visa-versa. When compared to an event-based observer pattern where objects send event notifications when changes are made performance can be worse. AngularJS plans on using the ECMAScript 6 (JavaScript) Object.observe() in place of dirty checking in future versions and falling back to their current dirty checking implementation for older browsers so over time this will become less of a concern.
- String-based templates do have some advantages over DOM-based
  - Imagine you had a table and wanted two table rows for each item you are iterating over. This is difficult to achieve in AngularJS because the ng-repeat directive is attached to a DOM element (a <tr> in this case). With Ember and Backbone the

iteration is not tied to a DOM element so this is not a problem.

- No main method, lots of magic
  - Since Angular has a run loop similar to game programming that listens for events such as changes in data and responds by data binding again it doesn't really have a main method where the code runs procedurally from top to bottom. This can make it more difficult to reason about and sometimes feel like a lot of "magic" is happening on your behalf. The Angular team recognizes this and says it's "good magic" and contends that main methods are very brittle and break easily because they depend so much on order of operations being maintained.
- Documentation
  - Although the community around AngularJS is big and eager to answer your questions, until recently there were public comments at the bottom of the official documentation and those explanations were frequently better than the actual documentation itself so you have to dig through all the comments to find the good ones. To be the fair, recently the documentation has been redone and gotten a lot better (and the public comments are gone) but in the past it could be infuriating or almost comical (depending on your mood) at times reading the old documentation.
- Syntax unnecessarily complex
  - The AngularJS APIs can be difficult, particularly for beginners, to understand because it uses "computer science" terms like transclusion and directive that are not immediately obvious.
- Difficult to learn
  - Because of the concept count, the syntax and the focus on dependency injection some people feel AngularJS is difficult to learn.
- One Company
  - Some argue that only having one company as the major contributor and backer for an open source project is a negative. In particular if that company is Google because they have a track record including Google Reader, Google Wave and FeedBurner which demonstrates their willingness to stop supporting or improving projects if they don't

align with the company's strategic goals.

- Lack of good support for partial views
  - Without AngularUI router, AngularJS has limited support for partial views. There is ng-include but only one ng-view directive is allowed per page which can be limiting when trying to compose a complex application from smaller pieces or UI.

Ember Pros

- Robust
  - Ember started from SproutCore and jQuery and trimmed down from there and it shows. It has a lot of functionality including some that people haven't even realized they need yet. As people build more and more complex single-page applications this depth of functionality is beginning to shine.
- Router
  - Ember's router is a full-state machine and doesn't just map URL's to functions. Ember's router supports nested views. Ember's overall architecture is based on the URL so it is difficult to make applications that break the back button.
- Partials
  - Ember has excellent support for UI composition and allows it to happen without breaking the back button.
- String templates can be more flexible than DOM
  - As mentioned earlier, imagine you had a table and wanted two table rows for each item you are iterating over. This is difficult to achieve in AngularJS but easy in Ember because the loop is not tied to the DOM element. Another advantage of string templates comes up in scenarios where you care about SEO and need to render your templates on the server for web crawlers. This can be much easier and more efficient with string-based templates because you don't need an in memory browser instance (because you don't need the DOM) to process the templates.
- Vision
  - Ember is similar to Angular in that it has a clear path to integrate future web standards in particular the new web components standards. Components fill the same needs as the web components standards.
- Productivity
  - The two-way data binding provided with Ember reduces a lot of boiler-plate code and allows for significantly less code to be written compared to a Backbone application.

- Components Easy
  - Ember's components are easier to write than their AngularJS counterpart: directives.
- jQuery +
  - Ember embraces and stands on the shoulders of jQuery. It doesn't try to dissuade you from using jQuery as AngularJS does which can make it easier to make the transition.
- More than just one company
  - There are 8 different companies represented on the Ember core team so it is not just dependent on one company. This diversified portfolio of contributors makes Ember less risky to adopt for many.
- Persistence
  - Ember Data is still in the works but when finished will be significantly more ambitious than the persistence support other frameworks currently provide. It will include an identity map on the client and support for hierarchical data syncing over the network.
- Rising in popularity
  - Although it is a distant second at this point they do seem to be chipping away at the lead AngularJS has in the framework race.

Ember Cons

- Can't find answers to questions
  - Stackoverflow.com has 9,183 Ember questions compared to AngularJS's 33,415 and Backbone's 14,524. In fairness, Ember has not been out as long. Exacerbating the situation however is the fact that many of the answers are outdated and incorrect because Ember released several beta versions and changed the API during that time.
- Documentation pretty but not deep
  - The current Ember documentation is well designed (pretty) and organized but does not go very deep into the corners of the framework. For example, the life cycle of a route seemed critical to understanding the framework but wasn't documented completely at the time this book was written.
- Persistence (Ember Data not finished)
  - Although ambitious, the reality is version 1 of Ember Data has still not shipped at the time of this writing and the prominent Ember projects like the open source Discourse (forum software) simply use jQuery .$ajax to make service calls.
- Lots of concepts
  - Given the robustness of the framework, decisions on which similar concept or part of the framework to use in a given situation come up frequently and it is not always obvious what the right choice should be to use the framework properly. For example when to use a view versus a partial versus a component needs a chart in the documentation.
- Terminology confusing in places
  - A route in most MVC frameworks is essentially a URL and what action or method it maps to but in Ember a route is an object responsible for loading the model. There are good reasons for this which come not from the framework's web heritage but it's SproutCore desktop application roots but it is still confusing particularly when first trying to understand the framework.

- Hard to learn
  - Because of how robust it is and its high concept count it can be the most difficult to learn.
- Immature
  - Ember was the last framework of those being discussed to reach version 1. Consequently, it also has the fewest examples of applications in production using the framework.
- Community
  - The Ember community is large but not as large as the other two frameworks being considered in this book although by some measures it has almost caught up to Backbone in popularity recently.
- Search-Engine Optimization (SEO) can be hurt
  - SEO concerns with these frameworks were outlined in detail in the AngularJS con list and won't be repeated here except to mention all frameworks have this concern.
- Getters and setters needed to access properties
  - Ember uses the observer pattern to do change tracking and enable two-way data binding. Consequently, getters and setters are required to access properties. As mentioned a few times previously it is common to forget this and run into hard to track down bugs.
- Data binding syntax
  - The data binding syntax is not as easy and clean with form inputs as it is in other frameworks. For example, this is an input: {{input value="http://www.facebook.com"}}. Essentially, you lose the angle brackets which would make the page read more like just plain HTML.

Backbone Pros

- Library not a Framework
  - Backbone and its dependency the utility library Underscore are about 1/3 the size of jQuery. Do you think twice about bringing jQuery into your project? Most likely not so it is just as inconsequential in my mind to bring in Backbone to help organize your existing jQuery code if for no other reasons. In summary, you aren't making a big commitment when including Backbone in your project and this is a big pro.
- Un-opinionated
  - Backbone has so few opinions on how to use it that it can be liberating for the developer that likes to roll his or her own framework and is comfortable with the JavaScript language.
- Easy to learn
  - Most people find Backbone very easy to learn because it is such a small library that you can just read the source code if you don't understand something.
- Builds on jQuery
  - Backbone views manage a DOM node and all the nodes under it usually using jQuery. The views have two methods initialize and render and is essentially just a wrapper around your normal jQuery code that helps organize it, making it is very comfortable and easy evolution for the jQuery developer.
- JQuery UI and other plugins are easily integrated
  - Because Backbone is just a light organizational wrapper as previously described it is very easy to use the thousands of existing jQuery plugins in your application.
- Great in brown field projects
  - Backbone's size and un-opinionated nature make it ideal to include and start using in an existing project without much risk.
- Most mature
  - Because it has been around the longest Backbone has the most impressive resume of

applications that use it as outlined previously including being shipped in the WordPress core and used in the Linked In mobile application.

- Stable
  - Being simple and mature Backbone has become extremely stable and has not undergone any major changes and has no road map to do so in the future.
- Documentation
  - The documentation is simple and minimal but complete. Of course this is easier given the breadth of the library but nonetheless a positive for Backbone.
- Underscore.js
  - As previously mentioned the only dependency of Backbone is Underscore.js and it is widely regarded as the best functional/utility library for JavaScript. Although other frameworks can use Underscore.js, the utility library works seamlessly with Backbone often being mixed in to the framework objects like Backbone.Collection.
- Templating Engine Flexibility
  - Backbone can easily be used with Handlebars.js, Dust.js or the micro-templating built into Underscore.js. This flexibility is useful, particularly since template libraries are frequently the source of performance problems in single-page applications.

Backbone Cons

- Not much there
  - Backbone can be too minimal for some. Although it prides itself on its size sometimes you may find yourself wishing for more features in the framework such as a more elaborate state-based router or more of a persistence story or at least some of the boiler-plate code you write over and over again to be abstracted into the core library.
- Library Not Framework
  - When doing more complex projects most people tend to use a framework with Backbone that adds missing functionality such as higher level view abstractions, modules etc... Marionette is the most popular alternative and makes Backbone into more of a framework. Thorax is another alternative and takes a much more declarative approach.
- Not prescriptive
  - As mentioned previously some developers find the un-opinionated stance of Backbone liberating but others want more guidance and help when developing their application.
- Can be difficult to learn
  - Since there is no "right way" to do a given thing in Backbone it sometimes takes substantial research and cognitive energy to figure out which way to head on a particular problem.
- Memory Management
  - It was very easy to create a Zombie (lost from your code but not from memory) view in early version of Backbone. Some of the Marionette ideas to make it easier to succeed have been adopted by Backbone but you are still handed a loaded gun and can "shoot your foot off."
- Extra elements needed in HTML
  - Because every Backbone view manages one root element in the DOM as well as every

element underneath the root element, it is common to need to create a container element to represent the view despite it not being needed by anything except the framework.

- Not evolving
  - There is not an ambitious road map for the future of Backbone because of its minimal philosophy so if you are looking to grow with a framework to higher and higher levels of abstraction and help in building your single-page applications this is not the framework for you.
- Getters and setters needed to access properties
  - Because Backbone uses the observer pattern to do change tracking so events can notify subscribers of changes; getters and setters are required to access properties. As mentioned a few times previously, it is common to forget this and consequently run into hard to track down bugs.
- No data binding
  - The lack of two-way data binding in Backbone is one of the major contributors to it always having more lines of code than the frameworks.
- Productivity Low
  - If you need to build something fast Backbone is probably not the framework for you. Backbone's verbose syntax, lack of data binding, getters and setters, and need for careful memory management all add up to create a C++ feel to development where you are really close to the metal but not very productive.
- Lack of complicated example applications
  - Backbone was the first to use a "todo" example and made it the canonical example for JavaScript MVC frameworks and libraries. For some reason the public open-source examples of Backbone applications have not evolved beyond this simplicity and the community could use some more substantial examples.

Conclusion

As I mentioned at the beginning of this chapter, no framework is perfect and choosing a framework is a series of trade-offs between the various strengths and weaknesses based on the needs of your project. This chapter should have made it clear what trade-offs you are accepting as you build your ambitious web application.

CHAPTER 11

RECOMMENDATIONS

Introduction

In this chapter I'm going to get specific about situations in which I would pick one framework over another and why. It's tempting to wish there was one framework that is always superior, but unfortunately it's not that simple.

Should I be using a framework?

If you haven't already, this is a good point to go back and skim over the "Should I be using a JavaScript Framework" [on my project] chapter and do an honest evaluation of whether you need one. In particular, developers (myself included) often have "shiny object syndrome" where they tend to always be interested in new technology but not always with business value as the driving force. Enough said on that but be sure that you can make a compelling business and/or technical argument for why a JavaScript MV* framework makes sense for your project.

Generations of Frameworks

It's important to understand the chronology of these libraries and frameworks to make a good choice. In the beginning, there was just JavaScript and almost everyone was afraid to use it because of the inconsistent DOM APIs across different browsers which resulted in lots of bugs. Next, jQuery came along and made DOM manipulation and AJAX calls less risky by abstracting away the differences into one consistent API. As developers began to write more and more JavaScript and jQuery to make their applications more innovative their code quickly became messier and difficult to manage. Backbone arrived and provided organization to that messy jQuery code and allowed a new generation of applications to be built in the browser by adding a thin layer on top of the familiar jQuery code developers were already writing.

Numerous Backbone clones emerged to create a first generation of libraries including Spine and JavaScriptMVC but Backbone survived as the favorite. Developers still found it difficult to be productive and longed for features such as data binding that they had used in Rich Internet Application (RIA) technologies like Flex and Silverlight. Knockout came along and provided data binding but didn't have any other features that developers needed to build single-page applications. At this point a second generation of more robust frameworks began to come out. AngularJS emerged with all the needed framework pieces in one box: data binding, routing, templating, persistence as well as a compelling testing story and a more declarative syntax. Ember was also built with similar features to AngularJS but a stronger router and more emphasis on URLs and not breaking the back button. Other notable frameworks from this second generation are CanJS and Durandal. Durandal showed innovation and promise but recently announced its plans to merge with AngularJS although Durandal will continue to be supported and evolve. CanJS is a slimmed down version of JavaScriptMVC from the first generation with more data binding support. CanJS just doesn't seem to have the community around it but remains popular within its niche.

It's interesting to note that jQuery was the one clear winner in the DOM manipulation library debate over Scriptaculous, MooTools, and many others. Similarly, the first generation of JavaScript MVC Frameworks also had one notable survivor in Backbone. After seeing AngularJS merge with Durandal, one can't help but wonder if Ember will be able to thrive in the long term. But most including myself believe it is still too early to make that pronouncement, particularly because of the strong allegiance in the Ruby community to Ember.

Brown-field or Green-field Project

Brown-field projects

A brown-field project involves problem spaces needing the development and deployment of new software systems in the immediate presence of existing (legacy) software applications/systems. This implies that any new software architecture must take into account and coexist with live software already in place.

Green-field projects

Contrastingly, a green-field project lacks any constraints imposed by prior work.

Backbone and other similar 1st generation libraries have a very small footprint and play well with existing jQuery code and plugins making them ideal for brown-field projects where a significant amount of jQuery code already exists.

2nd generation frameworks like AngularJS and Ember shine in green-field scenarios where there is a clean slate and developers often need to be more productive.

Fullstack considerations

Build and Deploy

It can be helpful to consider what server-side technology will be used on a given project because although all these client-side frameworks work with any server stack, some server technologies work better with certain frameworks, mostly because of how well they are integrated into the build tools and package system. For example, Ember has Ruby gems for pre-compiling assets and integrating your code into the asset pipeline so it plays nicer with a

Rails back end. Backbone was extracted from a Rails application so there is also good support in Rails for Backbone heavy applications. AngularJS comes out of Google where they mostly use the Java technology stack so AngularJS has great support in the Java community. Knockout and Durandal had a big following in the Microsoft .NET community from WPF and Silverlight developers who found the MVVM ideas in these frameworks familiar. .NET web developers now seem to be looking harder at Backbone and AngularJS. Again, all frameworks work with all server-side platforms. Some just might have less friction during build and deploy than others so be sure to weigh this as a factor in your decision.

Training

If you are training developers who are new to the technology, it is important to think about what technology they already have mastered, understand, and work in daily. More specifically, a Flex or Silverlight developer will feel much more at home with AngularJS or Ember than Backbone while a jQuery web developer will more easily grasp Backbone. Rails developers will find the Ember APIs familiar since the creators come from those experiences. These factors could significantly lower your ramp-up time and training costs for developers who are new to these technologies.

Lifespan

If an application will likely be mission critical to your business and will not be re-written for years, you may want to lean towards Ember and AngularJS since they seem to still be innovating new features and and have a clear path to supporting Web Components and Object.Observe from the ECMA 6 proposals.

Backbone: Safe Choice First Generation

If you want the safest choice that is guaranteed not to cause waves, then Backbone is clearly the smallest step and smallest commitment into this new world. With this safe choice, however, you will get lower productivity while developing your application and not have a clear path to the future as web standards evolve. You will also be making a conscious choice to ignore the second generation of technology evolution in this space. If you make this choice I would suggest using Marionette.js as well to eliminate the boilerplate code and help with memory management.

AngularJS: Safe Choice Second Generation

With a large community and Google backing it, AngularJS is the leader and safe choice of the second generation frameworks. They have dedicated employees at Google working to evolve the framework, and based on the design documents they've published for version 2.0, they seem to understand the weaknesses of the framework and have plans to improve it. If you make this choice, I would recommend using the AngularUI project's router to eliminate most of AngularJS's current limitations.

Ember: Robust

Although it may take your development team longer to understand Ember, it is robust enough to handle the harder problems as your application becomes more complex and your back button will work. If you make this choice, you should be aware of the possibility that AngularJS may become the de facto standard and the community around Ember could be significantly smaller, similar to prototype.js being embraced by the Ruby community but jQuery still emerging as a clear overall winner. The upside is if Ember Data meets its

ambitious goals you could have a more complete single-page application story to tell that includes persistence.

Third Generation

There will likely be a third-generation of frameworks to help us build client-side browser-based applications. The frameworks will allow developers to be more productive developing applications, ideally as productive as developers were in Flex and Silverlight in the past. The frameworks will allow richer interactions including making it easier to animate transitions between parts of the application. The frameworks may grow to encompass data access and persistence more seemlessly. The frameworks may just be the next versions of AngularJS and Ember, but they may be something entirely different. Welcome to exciting world of open source. I encourage you to embrace the new pace of innovation.

Conclusion

I've always prided myself on providing more realistic estimates for my software development work. I have learned from years of experience that things always seem to take longer than I thought and frequently two to three times longer. Despite this I somehow thought I could write this book in two to three months in the evenings. After seven months of working on this as my full time job I am writing the final words of this book and recording the final video interviews this week. I apparently need more experience at writing books so I can make better estimates.

It is my hope that you benefit from the months of hard work I put into this book and build awesome web applications with what you've learned.

Tweet This Book!

Please help Craig McKeachie @cmckeachie by spreading the word about this book on Twitter! The suggested tweet for this book is:

I just bought #jsframeguide, confidently choosing and quickly learning! I'm ready to build awesome web applications!

The suggested hashtag for this book is #jsframeguide

60-Minute Video Chat with the Author

A limited bonus offer (while availability lasts)

You've gone over all the material, but still have a few follow-up questions. Want to go straight to the source? I'm at your service. I'll simply ask you to purchase the team package of the book as compensation and will refund any prior purchases you have made. Email me at craig@funnyant.com to set up a time.