Contents

Prelude

Welcome to my (in-progress) book about the Backbone.js library for structuring JavaScript applications. It's released under a Creative Commons Attribution-NonCommercial-ShareAlike 3.0 Unported license meaning you can both grab a copy of the book for free or help to further improve it.

I'm very pleased to announce that this book will be out in physical form (once complete) via O'Reilly Media. Readers will have the option of purchasing the latest version in either print or a number of digital formats then or can grab a recent version from this repository.

Corrections to existing material are always welcome and I hope that together we can provide the community with an up-to-date resource that is of help. My extended thanks go out to Jeremy Ashkenas for creating Backbone.js and these members of the community for their assistance tweaking this project.

I hope you find this book helpful!

Introduction

Frank Lloyd Wright once said “You can’t make an architect. You can however open the doors and windows toward the light as you see it.” In this book, I hope to shed some light on how to improve the structure of your web applications, opening doors to what will hopefully be more maintainable, readable applications in your future.

The goal of all architecture is to build something well; in our case, to craft code that is enduring and delights both ourselves and the developers who will maintain our code long after we are gone. We all want our architecture to be simple, yet beautiful.

When writing a web application from scratch it can be easy to feel like you can get by simply relying on a DOM manipulation library (such as jQuery) and a handful of plugins. The challenge with this approach is that it doesn’t take long to get lost in a nested pile of callbacks and DOM elements without any real structure in place.

In short, you can end up with a pile of spaghetti code - code that is disorganized and difficult to follow. This type of code has no simple panacea, short of a rewrite that may end up costing both time and money to alleviate. Fortunately, there are ways to avoid this problem.

Modern JavaScript frameworks and libraries can bring structure and organization to your projects, establishing a maintainable foundation right from the start. They build on the trials and tribulations of developers who have had to work around callback chaos similar to that which you are facing now or may in the near future.

In “Developing Backbone.js Applications,” I and a number of other experienced authors will show you how to improve your web application structure using one such library - Backbone.js.

What Is MVC?

A number of modern JavaScript frameworks provide developers an easy path to organizing their code using variations of a pattern known as MVC (Model-View-Controller). MVC separates the concerns in an application into three parts:

- Models represent the domain-specific knowledge and data in an application. Think of this as being a ‘type’ of data you can model — like a User, Photo, or Todo note. Models can notify observers when their state changes.
- Views typically constitute the user interface in an application (e.g., markup and templates), but don’t have to be. They observe Models, but don’t directly communicate with them.

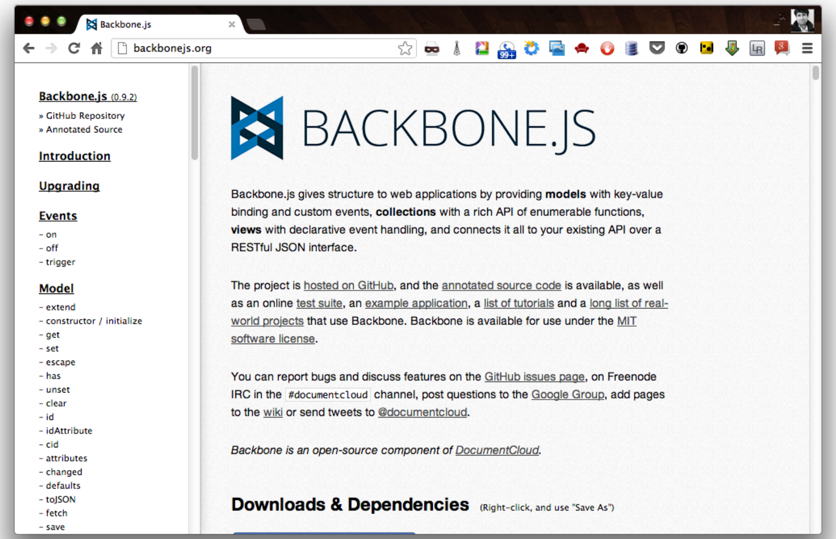
- Controllers handle input (e.g., clicks, user actions) and update Models.

Thus, in an MVC application, user input is acted upon by Controllers which update Models. Views observe Models and update the user interface when changes occur.

JavaScript MVC frameworks don't always strictly follow the above pattern. Some solutions (including Backbone.js) merge the responsibility of the Controller into the View, while other approaches add additional components into the mix.

For this reason we refer to such frameworks as following the MV* pattern; that is, you're likely to have a Model and a View, but a distinct Controller might not be present and other components may come into play.

What is Backbone.js?



Backbone.js (0.9.2)
» GitHub Repository
» Annotated Source

Introduction

Upgrading

Events
- on
- off
- trigger

Model
- extend
- constructor / initialize
- get
- set
- escape
- has
- unset
- clear
- id
- idAttribute
- cid
- attributes
- changed
- defaults
- toJSON
- fetch
- save

BACKBONE.JS

Backbone.js gives structure to web applications by providing **models** with key-value binding and custom events, **collections** with a rich API of enumerable functions, **views** with declarative event handling, and connects it all to your existing API over a RESTful JSON interface.

The project is hosted on GitHub, and the annotated source code is available, as well as an online test suite, an example application, a list of tutorials and a long list of real-world projects that use Backbone. Backbone is available for use under the MIT software license.

You can report bugs and discuss features on the GitHub issues page, on Freenode IRC in the `#documentcloud` channel, post questions to the Google Group, add pages to the wiki or send tweets to @documentcloud.

Backbone is an open-source component of DocumentCloud.

Downloads & Dependencies (Right-click, and use "Save As")

Backbone.js is a lightweight JavaScript library that adds structure to your client-side code. It makes it easy to manage and decouple concerns in your application, leaving you with code that is more maintainable in the long term.

Developers commonly use libraries like Backbone.js to create single-page applications (SPAs). SPAs are web applications that load into the browser and then react to data changes on the client side without requiring complete page refreshes from the server. Backbone.js is a mature, popular library at the time of writing and has both a large development community online as well as a wealth of plugins and extensions available that build upon it. It has been used to create non-trivial applications by companies such as Disqus, Walmart, SoundCloud and Foursquare.

When Do I Need A JavaScript MVC Framework?

When building a single-page application using JavaScript, whether it involves a complex user interface or is simply trying to reduce the number of HTTP requests required for new Views, you will likely find yourself inventing many of the pieces that make up an MV* framework.

At the outset, it isn't terribly difficult to write your own application framework that offers some opinionated way to avoid spaghetti code; however, to say that it is equally as trivial to write something as robust as Backbone would be a grossly incorrect assumption.

There's a lot more that goes into structuring an application than tying together a DOM manipulation library, templating, and routing. Mature MV* frameworks typically include not only the pieces you would find yourself writing, but also include solutions to problems you'll find yourself running into later on down the road. This is a time-saver that you shouldn't underestimate the value of.

So, where will you likely need an MV* framework and where won't you?

If you're writing an application where much of the heavy lifting for view rendering and data manipulation will be occurring in the browser, you may find a JavaScript MV* framework useful. Examples of applications that fall into this category are GMail and Google Docs.

These types of applications typically download a single payload containing all the scripts, stylesheets and markup users need for common tasks and then perform a lot of additional behavior in the background. For instance, it's trivial to switch between reading an email or document to writing one and a new page is never requested from the server.

If, however, you're building an application that still relies on the server for most of the heavy-lifting of page/view rendering and you're just using a little JavaScript or jQuery to make things more interactive, an MV* framework may be overkill. There certainly are complex Web applications where the partial rendering of views can be coupled with a single-page application effectively, but for everything else, you may find yourself better sticking to a simpler setup.

Maturity in software (framework) development isn't simply about how long a framework has been around. It's about how solid the framework is and more importantly how well it's evolved to fill its role. Has it become more effective at solving common problems? Does it continue to improve as developers build larger and more complex applications with it?

Why Consider Backbone.js?

Does the following describe you?:

"I want something flexible which offers a minimalist solution to separating concerns in my application. It should support a persistence layer and RESTful sync, Models, Views (with Controllers), event-driven communication, templating, and routing. It should be imperative, allowing Views to update themselves when Models change. I'd like some decisions about the architecture left up to me. Ideally, many large companies have used the solution to build non-trivial applications.

As I may be building something complex, I'd like there to be an active extension community around the framework that is already addressing issues I may run into down the road. Ideally, there are also scaffolding tools available for the solution."

If so, continue reading.

Backbone's main benefits, regardless of your target platform or device, include helping:

- Organize the structure to your application
- Simplify server-side persistence
- Decouple the DOM from your page's data
- Model data, views and routers in a succinct manner
- Provide DOM, model and collection synchronization

Setting Expectations

The goal of this book is to create an authoritative and centralized repository of information that can help those developing real-world apps with Backbone. If you come across a section or topic which you think could be improved or expanded on, please feel free to submit an issue (or better yet, a pull-request) on the book's GitHub site. It won't take long and you'll be helping other developers avoid the problems you ran into.

Topics will include MVC theory and how to build applications using Backbone's Models, Views, Collections, and Routers. I'll also be taking you through advanced topics like modular development with Backbone.js and AMD (via RequireJS), solutions to common problems like nested views, how to solve routing problems with Backbone and jQuery Mobile, and much more.

Fundamentals

In this first chapter, we're going to explore how frameworks like Backbone.js fit in the world of JavaScript application architecture. Classically, developers creating desktop and server-class applications have had a wealth of design patterns available for them to lean on, but it's only been in the past few years that such patterns have come to client-side development.

Before exploring any JavaScript frameworks that assist in structuring applications, it can be useful to gain a basic understanding of architectural design patterns.

MVC & Backbone.js

Design patterns are proven solutions to common development problems and can help guide us in adding more organization and structure to our applications. Patterns are also of great use as they reflect a set of practices which build upon the collective experience of skilled developers who have repeatedly solved similar problems.

In this section, we're going to explore the MVC (Model-View-Controller) pattern and how it applies to Backbone.js.

MVC

MVC is an architectural design pattern that encourages improved application organization through a separation of concerns. It enforces the isolation of business data (Models) from user interfaces (Views), with a third component (Controllers) traditionally managing logic, user-input and the coordination of models and views. The pattern was originally designed by Trygve Reenskaug while working on Smalltalk-80 (1979), where it was initially called Model-View-Controller-Editor. MVC was described in depth in "Design Patterns: Elements of Reusable Object-Oriented Software" (The "GoF" or "Gang of Four" book) in 1994, which played a role in popularizing its use.

Smalltalk-80 MVC

It's important to understand what the original MVC pattern was aiming to solve as it has changed quite heavily since the days of its origin. Back in the 70's, graphical user-interfaces were few and far between. An approach known as Separated Presentation began to be used as a means to make a clear division between domain objects which modeled concepts in the real world (e.g., a photo, a person) and the presentation objects which were rendered to the user's screen.

The Smalltalk-80 implementation of MVC took this concept further and had an objective of separating out the application logic from the user interface. The idea was that decoupling these parts of the application would also allow the reuse of models for other interfaces in the application. There are some interesting points worth noting about Smalltalk-80's MVC architecture:

- A Domain element was known as a Model and was ignorant of the user-interface (Views and Controllers)
- Presentation was taken care of by the View and the Controller, but there wasn't just a single view and controller. A View-Controller pair was required for each element being displayed on the screen and so there was no true separation between them
- The Controller's role in this pair was handling user input (such as key-presses and click events) and doing something sensible with them
- The Observer pattern was used to update the View whenever the Model changed

Developers are sometimes surprised when they learn that the Observer pattern (nowadays commonly implemented as a Publish/Subscribe system) was included as a part of MVC's architecture decades ago. In Smalltalk-80's MVC, the View and Controller both observe the Model: anytime the Model changes, the Views react. A simple example of this is an application backed by stock market data - for the application to show real-time information, any change to the data in its Model should result in the View being refreshed instantly.

Martin Fowler has done an excellent job of writing about the origins of MVC over the years and if you are interested in further historical information about Smalltalk-80's MVC, I recommend reading his work.

MVC Applied To The Web

The web heavily relies on the HTTP protocol, which is stateless. This means that there is not a constantly open connection between the browser and server; each request instantiates a new communication channel between the two. Once the request initiator (e.g. a browser) gets a response the connection is closed. This fact creates a completely different context when compared to the one of the operating systems on which many of the original MVC ideas were developed. The MVC implementation has to conform to the web context.

A typical server-side MVC implementation has one MVC stack layered behind a single point of entry. This single point of entry means that all HTTP requests, e.g., `http://www.example.com`, `http://www.example.com/whichever-page/`, etc., are routed by the server configuration to the same handler, independent of the URI.

At that point, there would be an implementation of the Front Controller pattern which analyzes HTTP requests and decides which class (Controller) and method (Action) to invoke in response to the request.

The selected Controller takes over and passes to and/or fetches data from the appropriate Model. After the Controller receives the data from the Model, it loads an appropriate View, injects the Model data into it, and returns the response to the browser.

For example, let say we have our blog on `www.example.com` and we want to edit an article (with `id=43`) and request `http://www.example.com/article/edit/43`:

On the server side, the Front Controller would analyze the URL and invoke the Article Controller (corresponding to the `/article/` part of the URI) and its Edit Action (corresponding to the `/edit/` part of the URI). Within the Action there would be a call to, lets say, the Articles model and its `Articles::getEntry(43)` method (43 corresponding to the `/43` at the end of the URI). This would return the blog article data from the database for edit. The Article Controller would then load the (`article/edit`) View which would include logic for injecting the article's data into a form suitable for editing its content, title, and other (meta) data. Finally, the resulting HTML response would be returned to the browser.

As you can imagine, a similar flow is necessary with POST requests after we press a save button in a form. The POST action URI would look like `/article/save/43`. The request would go through the same Controller, but this time the Save Action would be invoked (due to the `/save/` URI chunk), the Articles Model would save the edited article to the database with `Articles::saveEntry(43)`, and the browser would be redirected to the `/article/edit/43` URI for further editing.

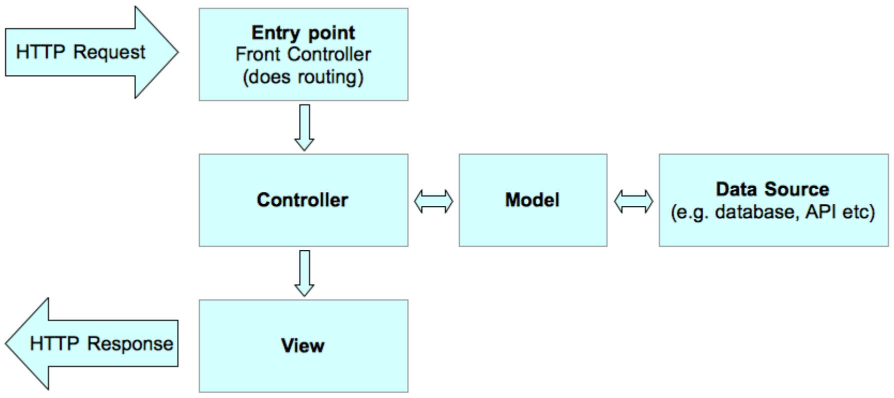
If the user requested `http://www.example.com/`:

On the server side the Front Controller would invoke the default Controller and Action; e.g., the Index Controller and its Index action. Within Index Action there would be a call to the Articles model and its `Articles::getLastEntries(10)` method which would return the last 10 blog posts. Afterwards, the Controller would load the blog/index View which would have basic logic for listing the blog posts.

We can see the larger picture of typical HTTP request lifecycle through the server side MVC in the picture below.

The Server receives an HTTP request and routes it through a single entry point. At that entry point, the Front Controller analyzes the request and based on it invokes an Action of the appropriate Controller. This process is called routing. The Action Model is asked to return and/or save submitted data. The Model communicates with the data source (e.g., database or API). Once the Model completes its work it returns data to the Controller which then loads the appropriate View. The View executes presentation logic (loops through articles and prints titles, content, etc.) using the supplied data. In the end, an HTTP response is returned to the browser.

The need for fast, complex, and responsive Ajax-powered web applications demands replication of a lot of this logic on the client side, dramatically increasing



the size and complexity of the code residing there. Eventually this has brought us to the point where we need MVC (or a similar architecture) implemented on the client side to better structure the code and make it easier to maintain and further extend during the application life-cycle.

And, of course, JavaScript and browsers create another context to which we must adjust the traditional MVC paradigm.

MVC In The Browser

In complex JavaScript single-page web applications (SPA), all application responses (e.g., UI updates) to user inputs are done seamlessly on the client-side. Data fetching and persistence (e.g., saving to a database on a server) are done with Ajax in the background. For silky, slick, and smooth user experiences, the code powering these interacions needs to be well thought out.

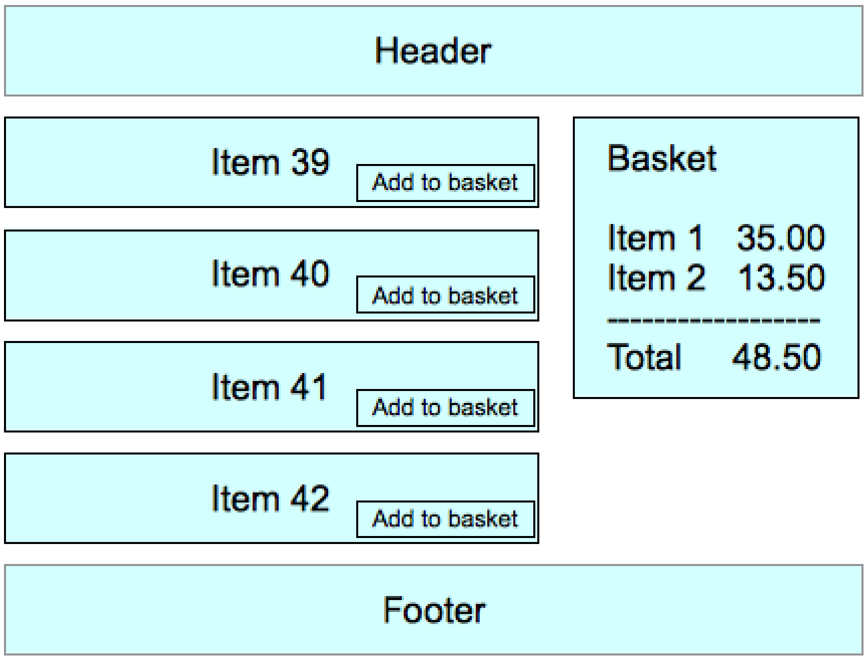
Through evolution, trial and error, and a lot of spaghetti (and not so spaghetti-like) code, JavaScript developers have, in the end, harnessed the ideas of the traditional MVC paradigm. This has led to the development of a number of JavaScript MVC frameworks, including Ember.js, JavaScriptMVC, and of course Backbone.js.

The problem

A typical page in an SPA consists of small elements representing logical entities. These entities belong to specific data domains that should be represented in a particular way on the page.

A good example is a basket in an e-commerce web application which can have a items added to it. This basket might be presented to the user in a box in the top right corner of the page (see the picture).

The basket and its data are presented in HTML. The data and its associated view in HTML changes over time. There was a time when we used jQuery (or a



similar DOM manipulation library) and a bunch of Ajax calls and callbacks to keep the two in sync. That often produced code that was not well structured or easy to maintain. Bugs were frequent and perhaps even unavoidable.

Eventually, an elegant way to handle it was brought to the client side throught JavaScript MVC libraries.

Using MVC, data is handled with a Model and its HTML presentation with a View. When the Model changes the View is updated and vice versa. A Controller manages this synchronization. It sends update commands both ways: to the View to update itself when the Model changes (e.g., synchronization with the database) and to the Model when the View changes (e.g., new items dropped into the basket). This results in a better separation of concerns and improved code structure.

Simple JavaScript MVC Implementation

Let's see a simple implementation of the MVC pattern and its usage to clarify some concepts - we're going to call our little library Cranium.js.

To simplify a bit we will rely on Underscore for inheritance and templating (similar to Backbone).

Event System At the heart of our JavaScript MVC implementation is an `Event` system (object) based on the Publisher-Subscriber Pattern which makes it possible for MVC components to intercommunicate in an elegant, decoupled manner. Subscribers 'listen' for specific events of interest and react when Publishers broadcast these events.

`Event` is inherited by the View and Model components. That way each of them can inform other components that event of the interest has occurred.

```javascript
// cranium.js - Cranium.Events

var Cranium = Cranium || {};

// Set DOM selection utility
var $ = document.querySelector.bind(document) || this.jQuery || this.Zepto;

// Mix in to any object in order to provide it with custom events.
var Events = Cranium.Events = {
  // Keeps list of events and associated listeners
  channels: {},

  // Counter
  eventNumber: 0,

  // Announce events and passes data to the listeners;
  trigger: function (events, data) {
    for (var topic in Cranium.Events.channels){
      if (Cranium.Events.channels.hasOwnProperty(topic)) {
        if (topic.split("-")[0] == events){
          Cranium.Events.channels[topic](data) !== false || delete Cranium.Events.channels[t
        }
      }
    }
  },
  // Registers an event type and its listener
  on: function (events, callback) {
    Cranium.Events.channels[events + --Cranium.Events.eventNumber] = callback;
  },
  // Unregisters an event type and its listener
  off: function(topic) {
    delete Cranium.Events.channels[topic];
  }
};
```

The Event system makes it possible for:

- a View to notify its subscribers of user interaction (e.g., clicks or input in a form), to update/re-render its presentation, etc.
- a Model whose data has changed to notify its Subscribers to update themselves (e.g., view to re-render to show accurate/updated data), etc.

Models Models manage the (domain-specific) data for an application. They are concerned with neither the user-interface nor presentation layers, but instead represent structured data that an application may require. When a model changes (e.g when it is updated), it will typically notify its observers (Subscribers) that a change has occurred so that they may react accordingly.

Let's see a simple implementation of the Model:

```
// cranium.js - Cranium.Model

// Attributes represents data, model's properties.
// These are to be passed at Model instantiation.
// Also we are creating id for each Model instance
// so that it can identify itself (e.g. on chage
// announcements)
var Model = Cranium.Model = function (attributes) {
    this.id = _.uniqueId('model');
    this.attributes = attributes || {};
};

// Getter (accessor) method;
// returns named data item
Cranium.Model.prototype.get = function(attrName) {
    return this.attributes[attrName];
};

// Setter (mutator) method;
// Set/mix in into model mapped data (e.g.{name: "John"})
// and publishes the change event
Cranium.Model.prototype.set = function(attrs){
    if (_.isObject(attrs)) {
      _.extend(this.attributes, attrs);
      this.change(this.attributes);
    }
    return this;
};

// Returns clone of the Models data object
// (used for view template rendering)
Cranium.Model.prototype.toJSON = function(options) {
    return _.clone(this.attributes);
};

// Helper function that announces changes to the Model
// and passes the new data
Cranium.Model.prototype.change = function(attrs){
    this.trigger(this.id + 'update', attrs);
};

// Mix in Event system
_.extend(Cranium.Model.prototype, Cranium.Events);
```

Views Views are a visual representation of models that present a filtered view of their current state. A view typically observes a model and is notified when the model changes, allowing the view to update itself accordingly. Design pattern literature commonly refers to views as 'dumb', given that their knowledge of models and controllers in an application is limited.

Let's explore Views a little further using a simple JavaScript example:

```
// DOM View
var View = Cranium.View = function (options) {
    // Mix in options object (e.g extending functionallity)
  _.extend(this, options);
  this.id = _.uniqueId('view');
};

// Mix in Event system
_.extend(Cranium.View.prototype, Cranium.Events);
```

Controllers Controllers are an intermediary between models and views which are classically responsible for two tasks:

- they update the view when the model changes
- they update the model when the user manipulates the view

```
// cranium.js - Cranium.Controller

// Controller tying together a model and view
var Controller = Cranium.Controller = function(options){
    // Mix in options object (e.g extending functionallity)
  _.extend(this, options);
  this.id = _.uniqueId('controller');
  var parts, selector, eventType;
  // Parses Events object passed during the definition of the
    // controller and maps it to the defined method to handle it;
  if(this.events){
    _.each(this.events, function(method, eventName){
      parts = eventName.split('.');
      selector = parts[0];
      eventType = parts[1];
      $(selector)['on' + eventType] = this[method];
    }.bind(this));
  }
};
```

Practical Usage HTML template for the primer that follows:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title></title>
  <meta name="description" content="">
</head>
<body>
<div id="todo">
</div>
    <script type="text/template" class="todo-template">
    <div>
        <input id="todo_complete" type="checkbox" <%= completed %>>
        <%= title %>
    </div>
    </script>
    <script src="underscore-min.js"></script>
  <script src="cranium.js"></script>
  <script src="example.js"></script>
</body>
</html>
```

Cranium.js usage:

```
// example.js - usage of Cranium MVC

// And todo instance
var todo1 = new Cranium.Model({
    title: "",
    completed: ""
});

console.log("First todo title - nothing set: " + todo1.get('title'));
todo1.set({title: "Do something"});
console.log("Its changed now: " + todo1.get('title'));
''
// View instance
var todoView = new Cranium.View({
    // DOM element selector
  el: '#todo',

  // Todo template; Underscore temlating used
  template: _.template($('.todo-template').innerHTML),

  init: function (model) {
        this.render( model.toJSON() );

    this.on(model.id + 'update', this.render.bind(this));
  },
  render: function (data) {
    console.log("View about to render.");
        $(this.el).innerHTML = this.template( data );
  }
});

var todoController = new Cranium.Controller({
  // Specify the model to update
  model: todo1,

  // and the view to observe this model
  view:  todoView,

  events: {
    "#todo.click" : "toggleComplete"
  },

  // Initialize everything
  initialize: function () {
    this.view.init(this.model);
    return this;
  },
  // Toggles the value of the todo in the Model
  toggleComplete: function () {
    var completed = todoController.model.get('completed');
    console.log("Todo old 'completed' value?", completed);
    todoController.model.set({ completed: (!completed) ? 'checked': '' });
    console.log("Todo new 'completed' value?", todoController.model.get('completed'));
    return this;
  }
});


// Let's kick start things off
todoController.initialize();

todo1.set({ title: "Due to this change Model will notify View and it will re-render"});
```

Imlementation Specifics

Notes on Model

- The built-in capabilities of models vary across frameworks; however, it's common for them to support validation of attributes, where attributes represent the properties of the model, such as a model identifier.
- When using models in real-world applications we generally also need a way of persisting models. Persistence allows us to edit and update models with the knowledge that their most recent states will be saved somewhere, for example in a web browser's localStorage data-store or synchronized with a database.
- A model may also have single or multiple views observing it. Depending on the requirements, a developer might create a single view displaying all Model attributes, or they might create three separate views displaying different attributes. The important point is that the model doesn't care how these views are organized, it simply announces updates to its data as necessary. It does so through a system like the Event System in our Cranium Framework.
- It is not uncommon for modern MVC/MV* frameworks to provide a means of grouping models together. In Backbone, these groups are called "Collections." Managing models in groups allows us to write application logic based on notifications from the group when a model within the group changes. This avoids the need to manually observe individual model instances. We'll see this in action later in the book.
- If you read older texts on MVC, you may come across a description of models as also managing application "state". In JavaScript applications state has a specific meaning, typically referring to the current state of a view or sub-view on a user's screen at a fixed time. State is a topic which

is regularly discussed when looking at Single-page applications, where the concept of state needs to be simulated.

Notes on View

- Users interact with views, which usually means reading and editing model data. For example, in todo application example we will examine, todo model viewing happens in the user interface in the list of all todo items. Within it, each todo is rendered with its title and completed checkbox. Model editing is done through an "edit" view where a user who has selected a specific todo edits its title in a form.
- We define a `render()` utility within our view which is responsible for rendering the contents of the `Model` using a JavaScript templating engine (provided by Underscore.js) and updating the contents of our view, referenced by `el`.
- We then add our `render()` callback as one of `Model` subscribers, so that through the Event System the view can be triggered to update when the model changes.
- You may wonder where user interaction comes into play here. When users click on a todo element within the view, it's not the view's responsibility to know what to do next. A Controller makes this decision. In our implementation, this is achieved by adding an event listener to a todo element which delegates handling of the click back to the controller, passing the model information along with it in case it's needed. The benefit of this architecture is that each component plays its own separate role in making the application function as needed.

Templating

In the context of JavaScript frameworks that support MVC/MV*, it is worth looking more closely at JavaScript templating and its relationship to Views.

It has long been considered bad practice (and computationally expensive) to manually create large blocks of HTML markup in-memory through string concatenation. Developers using this technique often find themselves iterating through their data, wrapping it in nested divs and using outdated techniques such as `document.write` to inject the 'template' into the DOM. This approach often means keeping scripted markup inline with standard markup, which can quickly become difficult to read and maintain, especially when building large applications.

JavaScript templating libraries (such as Handlebars.js or Mustache) are often used to define templates for views as HTML markup containing template variables. These template blocks can be either stored externally or within script tags with a custom type (e.g 'text/template'). Variables are delimited using a variable syntax (e.g {{title}}). Javascript template libraries typically accept data in JSON, and the grunt work of populating templates with data is taken care of by the framework itself. This has a several benefits, particularly when opting to store templates externally as enables applications to load templates dynamically on an as-needed basis.

Let's compare two examples of HTML templates. One is implemented using the popular Handlebars.js library, and the other uses Underscore's 'microtemplates'.

Handlebars.js:

```
<div class="view">
  <input class="toggle" type="checkbox" {{#if completed}} "checked" {{/if}}>
  <label>{{title}}</label>
  <button class="destroy"></button>
</div>
<input class="edit" value="{{title}}">
```

Underscore.js Microtemplates:

```
<div class="view">
  <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
  <label><%= title %></label>
  <button class="destroy"></button>
</div>
<input class="edit" value="<%= title %>">
```

You may also use double curly brackets (i.e `{{}}`) (or any other tag you feel comfortable with) in Microtemplates. In the case of curly brackets, this can be done by setting the Underscore `templateSettings` attribute as follows:

```
_.templateSettings = { interpolate : /\{\{(.+?)\}\}/g };
```

A note on navigation and state

It is also worth noting that in classical web development, navigating between independent views required the use of a page refresh. In single-page JavaScript applications, however, once data is fetched from a server via Ajax, it can be dynamically rendered in a new view within the same page. Since this doesn't automatically update the URL, the role of navigation thus falls to a "router", which assists in managing application state (e.g., allowing users to bookmark a particular view they have navigated to). As routers are neither a part of MVC nor present in every MVC-like framework, I will not be going into them in greater detail in this section.

Notes on Controller In our Todo application, a controller would be responsible for handling changes the user made in the edit view for a particular todo, updating a specific todo model when a user has finished editing.

It's with controllers that most JavaScript MVC frameworks depart from the traditional interpretation of the MVC pattern. The reasons for this vary, but in my opinion, Javascript framework authors likely initially looked at server-side interpretations of MVC (such as Ruby on Rails), realized that that approach didn't translate 1:1 on the client-side, and so re-interpreted the C in MVC to solve their state management problem. This was a clever approach, but it can make it hard for developers coming to MVC for the first time to understand both the classical MVC pattern and the "proper" role of controllers in other JavaScript frameworks.

So does Backbone.js have Controllers? Not really. Backbone's Views typically contain "controller" logic, and Routers are used to help manage application state, but neither are true Controllers according to classical MVC.

In this respect, contrary to what might be mentioned in the official documentation or in blog posts, Backbone isn't truly an MVC framework. It's in fact better to see it a member of the MV* family which approaches architecture in its own way. There is of course nothing wrong with this, but it is important to distinguish between classical MVC and MV* should you be relying on discussions of MVC to help with your Backbone projects.

What does MVC give us?

To summarize, the separation of concerns in MVC facilitates modularization of an application's functionality and enables:

- Easier overall maintenance. When updates need to be made to the application it is clear whether the changes are data-centric, meaning changes to models and possibly controllers, or merely visual, meaning changes to views.
- Decoupling models and views means that it's straight-forward to write unit tests for business logic
- Duplication of low-level model and controller code is eliminated across the application
- Depending on the size of the application and separation of roles, this modularity allows developers responsible for core logic and developers working on the user-interfaces to work simultaneously

Delving deeper

Right now, you likely have a basic understanding of what the MVC pattern provides, but for the curious, we'll explore it a little further.

The GoF (Gang of Four) do not refer to MVC as a design pattern, but rather consider it a “set of classes to build a user interface.” In their view, it’s actually a variation of three other classical design patterns: the Observer (Pub/Sub), Strategy, and Composite patterns. Depending on how MVC has been implemented in a framework, it may also use the Factory and Decorator patterns. I’ve covered some of these patterns in my other free book, “JavaScript Design Patterns For Beginners” if you would like to read about them further.

As we’ve discussed, models represent application data, while views handle what the user is presented on screen. As such, MVC relies on Publish/Subscribe for some of its core communication (something that surprisingly isn’t covered in many articles about the MVC pattern). When a model is changed it “publishes” to the rest of the application that it has been updated. The “subscriber,” generally a Controller, then updates the view accordingly. The observer-viewer nature of this relationship is what facilitates multiple views being attached to the same model.

For developers interested in knowing more about the decoupled nature of MVC (once again, depending on the implementation), one of the goals of the pattern is to help define one-to-many relationships between a topic and its observers. When a topic changes, its observers are updated. Views and controllers have a slightly different relationship. Controllers facilitate views’ responses to different user input and are an example of the Strategy pattern.

Summary

Having reviewed the classical MVC pattern, you should now understand how it allows developers to cleanly separate concerns in an application. You should also now appreciate how JavaScript MVC frameworks may differ in their interpretation of MVC, and how they share some of the fundamental concepts of the original pattern.

When reviewing a new JavaScript MVC/MV* framework, remember - it can be useful to step back and consider how it’s opted to approach Models, Views, Controllers or other alternatives, as this can better help you understand how the framework is intended to be used.

Further reading

If you are interested in learning more about the variation of MVC which Backbone.js is better categorized under, please see the MVP (Model-View-Presenter) section in the appendix.

Fast facts

Backbone.js

- Core components: Model, View, Collection, Router. Enforces its own flavor of MV*
- Used by large companies such as SoundCloud and Foursquare to build non-trivial applications
- Event-driven communication between views and models. As we'll see, it's relatively straight-forward to add event listeners to any attribute in a model, giving developers fine-grained control over what changes in the view
- Supports data bindings through manual events or a separate Key-value observing (KVO) library
- Support for RESTful interfaces out of the box, so models can be easily tied to a backend
- Extensive eventing system. It's trivial to add support for pub/sub in Backbone
- Prototypes are instantiated with the `new` keyword, which some developers prefer
- Agnostic about templating frameworks, however Underscore's micro-templating is available by default. Backbone works well with libraries like Handlebars
- Doesn't support deeply nested models, though there are Backbone plugins such as Backbone-relational which can help
- Clear and flexible conventions for structuring applications. Backbone doesn't force usage of all of its components and can work with only those needed.

The Internals

In this section, you'll learn the essentials of Backbone's models, views, collections and routers, as well as about using namespacing to organize your code. This isn't meant as a replacement for the official documentation, but it will help you understand many of the core concepts behind Backbone before you start building applications with it.

- Models
- Collections
- Routers
- Views
- Dependencies
- Namespacing

Models

Backbone models contain interactive data for an application as well as the logic around this data. For example, we can use a model to represent the concept of a todo item including its attributes like title (todo content) and completed (current state of the todo).

Models can be created by extending `Backbone.Model` as follows:

```
var Todo = Backbone.Model.extend({});

// We can then create our own concrete instance of a (Todo) model
// with no values at all:
var todo1 = new Todo();
console.log(todo1);

// or with some arbitrary data:
var todo2 = new Todo({
  title: 'Check attributes property of the both model instances in the console.',
  completed: true
});
console.log(todo2);
```

Initialization The `initialize()` method is called when a new instance of a model is created. Its use is optional, however you'll see why it's good practice to use it below.

```
var Todo = Backbone.Model.extend({
  initialize: function(){
      console.log('This model has been initialized.');
  }
});

var myTodo = new Todo();
```

Default values

There are times when you want your model to have a set of default values (e.g. in a scenario where a complete set of data isn't provided by the user). This can be set using a property called `defaults` in your model.

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Now we can create our concrete instance of the model
// with default values as follows:
var todo1 = new Todo();
console.log(todo1);

// Or we could instantiate it with some of the attributes (e.g with custom title):
var todo2 = new Todo({
  title: 'Check attributes property of the logged models in the console.'
});
console.log(todo2);

// Or with all of the (default) attributes:
var todo3 = new Todo({
  title: 'This todo is done, so take no action on this one.',
  completed: true
});
console.log(todo3);
```

Getters & Setters   Model.get()

`Model.get()` provides easy access to a model's attributes. All attributes, regardless if default ones were passed through to the model on instantiation, are available for retrieval.

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

var todo1 = new Todo();
console.log(todo1.get('title')); // empty string
console.log(todo1.get('completed')); // false

var todo2 = new Todo({
  title: "Retrieved with models get() method.",
  completed: true
});
console.log(todo2.get('title')); // Retrieved with models get() method.
console.log(todo2.get('completed')); // true
```

If you need to read or clone all of a model's data attributes use its `toJSON` method. Despite the name it doesn't return a JSON string but a copy of the attributes as an object. ("toJSON" is part of the JSON.stringify specification. Passing an object with a toJSON method makes it stringify the return value of that method instead of the object itself.)

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

var todo1 = new Todo();
var todo1Attributes = todo1.toJSON();
// Following logs: {"title":"","completed":false}
console.log(todo1Attributes);

var todo2 = new Todo({
  title: "Try these examples and check results in console.",
  completed: true
});
// logs: {"title":"Try examples and check results in console.","completed":true}
console.log(todo2.toJSON());
```

Model.set()

`Model.set()` allows us to pass attributes into an instance of our model. Attributes can either be set during initialization or at any time afterwards. Backbone uses Model.set() to know when to broadcast that a model's data has changed.

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  }
});

// Setting the value of attributes via instantiation
var myTodo = new Todo({
  title: "Set through instantiation."
});
console.log('Todo title: ' + myTodo.get('title'));
console.log('Completed: ' + myTodo.get('completed'));

// Set single attribute value at the time through Model.set():
myTodo.set("title", "Title attribute set through Model.set().");
console.log('Todo title: ' + myTodo.get('title'));
console.log('Completed: ' + myTodo.get('completed'));

// Set map of attributes through Model.set():
myTodo.set({
  title: "Both attributes set through Model.set().",
  completed: true
});
console.log('Todo title: ' + myTodo.get('title'));
console.log('Completed: ' + myTodo.get('completed'));
```

Direct access

If you really need to access the attributes in a model's instance directly, there is `Model.attributes`. But remember it is best practice to use Model.get(), Model.set() or direct instantiation as explained above.

Listening for changes to your model Any and all of the attributes in a Backbone model can have listeners bound to them which detect when their values change. Listeners can be added to the `initialize()` function:

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  },
  initialize: function(){
    console.log('This model has been initialized.');
    this.on('change', function(){
        console.log('- Values for this model have changed.');
    });
  }
});

var myTodo = new Todo();

myTodo.set('title', 'On each change of attribute values listener is triggered.');
console.log('Title has changed: ' + myTodo.get('title'));

myTodo.set('completed', true);
```

```
console.log('Completed has changed: ' + myTodo.get('completed'));

myTodo.set({
  title: 'Listener is triggered for each change, not for change of the each attribute.',
  'complete': true
});
```

In the following example, we log a message whenever a specific attribute (the title of our Todo model) is altered.

```
var Todo = Backbone.Model.extend({
  // Default todo attribute values
  defaults: {
    title: '',
    completed: false
  },
  initialize: function(){
    console.log('This model has been initialized.');
    this.on('change:title', function(){
        console.log('Title value for this model have changed.');
    });
  },
  setTitle: function(newTitle){
    this.set({ title: newTitle });
  }
});

var myTodo = new Todo();

// Following changes trigger the listener:
myTodo.set('title', 'Check what\'s logged.');
myTodo.setTitle('Go fishing on Sunday.');

// But, this change type is not observed, so no listener is triggered:
myTodo.set('completed', true);
console.log('Todo set as completed: ' + myTodo.get('completed'));
```

Validation Backbone supports model validation through `Model.validate()`, which allows checking the attribute values for a model prior to them being set.

Validation functions can be as simple or complex as necessary. If the attributes provided are valid, nothing should be returned from `.validate()`. If they are invalid, a custom error can be returned instead.

A basic example for validation can be seen below:

```
var Todo = Backbone.Model.extend({
  validate: function(attribs){
    if(attribs.title === undefined){
        return "Remember to set a title for your todo.";
    }
  },

  initialize: function(){
    console.log('This model has been initialized.');
    this.on("invalid", function(model, error){
        console.log(error);
    });
  }
});

var myTodo = new Todo();
myTodo.set('completed', false); // logs: Remember to set a title for your todo.
```

Note: Backbone passes the `attributes` object (attribs param in above example) by shallow copy to the `validate` function using the Underscore `_.extend` method. This means that it is not possible to change any Number, String or Boolean attribute but it *is* possible to change attributes of objects because they are passed by reference. As shallow copy doesn't copy objects by implicitly copying them, but rather, by reference, one can change the attributes on those objects.

An example of this (by @fivetanley) is available here.

Views

Views in Backbone don't contain the markup for your application, but rather they are there to support models by defining the logic for how they should be represented to the user. This is usually achieved using JavaScript templating (e.g. Mustache, jQuery-tmpl, etc.). A view's `render()` function can be bound to a model's `change()` event, allowing the view to always be up to date without requiring a full page refresh.

Creating new views Similar to the previous sections, creating a new view is relatively straight-forward. To create a new View, simply extend `Backbone.View`. I'll explain this code in detail below:

```
var TodoView = Backbone.View.extend({

  tagName:  'li',
  // Cache the template function for a single item.
  todoTpl: _.template( $('#item-template').html() ),

  events: {
    'dblclick label': 'edit',
    'keypress .edit': 'updateOnEnter',
    'blur .edit':   'close'
  },

  // Re-render the titles of the todo item.
  render: function() {
    this.$el.html( this.todoTpl( this.model.toJSON() ) );
    this.input = this.$('.edit');
    return this;
  },

  edit: function() {
    // executed when todo label is double clicked
  },

  close: function() {
    // executed when todo loses focus
  },

  updateOnEnter: function( e ) {
    // executed on each keypress when in todo edit mode,
    // but we'll wait for enter to get in action
  }
});

var todoView = new TodoView();

// logs reference to a DOM element that cooresponds to the view instance
console.log(todoView.el);
```

What is `el`? `el` is basically a reference to a DOM element and all views must have one. It allows for all of the contents of a view to be inserted into the DOM at once, which makes for faster rendering because the browser performs the minimum required reflows and repaints.

There are two ways to attach a DOM element to a view: a new element is created for the view and added manually by the developer or the element already exists in the page.

If you want to create a new element for your view, set any combination of the following view's properties: `tagName`, `id` and `className`. A new element will be created for you by the framework and a reference to it will be available at the `el` property. If nothing is specified `el` defaults to `div`.

```
var TodosView = Backbone.View.extend({
  tagName: 'ul', // required, but defaults to 'div' if not set
  className: 'container', // optional, you can assign multiple classes to this property lik
  id: 'todos', // optional
});

var todosView = new TodosView();
console.log(todosView.el);
```

The above code creates the `DOMElement` below but doesn't append it to the DOM.

```
<ul id="todos" class="container"></ul>
```

If the element already exists in the page, you can set `el` as a CSS selector that matches the element.

```
el: '#footer'
```

Understanding `render()`

`render()` is an optional function that defines the logic for rendering a template. We'll use Underscore's micro-templating in these examples, but remember you can use other templating frameworks if you prefer.

The `_.template` method in Underscore compiles JavaScript templates into functions which can be evaluated for rendering. In the above view, I'm passing the markup from a template with id `item-template` to `_.template()` to be compiled. Next, I set the html of the `el` DOM element to the output of processing a JSON version of the model associated with the view through the compiled template.

Presto! This populates the template, giving you a data-complete set of markup in just a few short lines of code.

The `events` attribute

The Backbone `events` attribute allows us to attach event listeners to either custom selectors, or directly to `el` if no selector is provided. An event takes the form `{'eventName selector': 'callbackFunction'}` and a number of DOM event-types are supported, including `click`, `submit`, `mouseover`, `dblclick` and more.

What isn't instantly obvious is that under the bonnet, Backbone uses jQuery's `.delegate()` to provide instant support for event delegation but goes a little further, extending it so that `this` always refers to the current view object. The only thing to really keep in mind is that any string callback supplied to the events attribute must have a corresponding function with the same name within the scope of your view.

Collections

Collections are sets of Models and are created by extending `Backbone.Collection`.

Normally, when creating a collection you'll also want to pass through a property specifying the model that your collection will contain, as well as any instance properties required.

In the following example, we create a TodoCollection that will contain our Todo models:

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  localStorage: new Store('todos-backbone')
});

var myTodo = new Todo({title:'Read the whole book', id: 2});

// pass array of models on collection instantiation
var todos = new TodosCollection([myTodo]);
console.log("Collection size: " + todos.length);

// Collection's convenience method used to create
// new model instance within collection itself.
todos.create({title:'Try out code examples', id: 48});
console.log("Collection size: " + todos.length);
```

Getters and Setters

There are a few different ways to retrieve a model from a collection. The most straight-forward is to use `Collection.get()` which accepts a single id as follows:

```
// extends on previous example
```

```
var todo2 = todos.get(2);

// Models, as objects, are passed by reference
console.log(todo2 === myTodo);
```

Internally `Backbone.Collection` sets an array of models enumerated by their `id` property, if model instances happen to have one. Once `collection.get(id)` is called this array is checked for existence of the model instance with the corresponding `id`.

Sometimes you may also want to get a model based on its client id. The client id is a property that Backbone automatically assigns models that have not yet been saved. You can get a model's client id from its `.cid` property.

```
// extends on previous examples

var todoCid = todos.get(todo2.cid);

// As mentioned in previous example,
// models are passed by reference
console.log(todoCid === myTodo);
```

Backbone Collections don't have setters as such, but do support adding new models via `.add()` and removing models via `.remove()`.

```
var Todo = Backbone.Model.extend({
  defaults: {
    title: '',
    completed: false
  }
});

var TodosCollection = Backbone.Collection.extend({
  model: Todo,
  localStorage: new Store('todos-backbone')
});

var a = new Todo({ title: 'Go to Jamaica.'}),
    b = new Todo({ title: 'Go to China.'}),
    c = new Todo({ title: 'Go to Disneyland.'});

var todos = new TodosCollection([a,b]);
console.log("Collection size: " + todos.length);

todos.add(c);
```

```
console.log("Collection size: " + todos.length);

todos.remove([a,b]);
console.log("Collection size: " + todos.length);

todos.remove(c);
console.log("Collection size: " + todos.length);
```

Listening for events

As collections represent a group of items, we're also able to listen for `add` and `remove` events for when new models are added or removed from the collection. Here's an example:

```
var TodosCollection = new Backbone.Collection();

TodosCollection.on("add", function(todo) {
  console.log("I should " + todo.get("title") + ". Have I done it before? "  + (todo.get("co
});

TodosCollection.add([
  { title: 'go to Jamaica.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Disneyland.', completed: true }
]);
```

In addition, we're able to bind a `change` event to listen for changes to models in the collection.

```
var TodosCollection = new Backbone.Collection();

TodosCollection.on("change:title", function(model) {
    console.log("Changed my mind where I should go, " + model.get('title'));
});

TodosCollection.add([
  { title: 'go to Jamaica.', completed: false, id: 3 },
]);

var myTodo = TodosCollection.get(3);

myTodo.set('title', 'go fishing');
```

Fetching models from the server

`Collections.fetch()` retrieves a default set of models from the server in the form of a JSON array. When this data returns, the current collection's contents will be replaced with the contents of the array.

```
var TodosCollection = new Backbone.Collection;
TodosCollection.url = '/todos';
TodosCollection.fetch();
```

During configuration, Backbone sets a variable to denote if extended HTTP methods are supported by the server. Another setting controls if the server understands the correct MIME type for JSON:

```
Backbone.emulateHTTP = false;
Backbone.emulateJSON = false;
```

The Backbone.sync method that uses these values is actually an integral part of Backbone.js. A jQuery-like ajax method is assumed, so HTTP parameters are organised based on jQuery's API. Searching through the code for calls to the sync method show it's used whenever a model is saved, fetched, or deleted (destroyed).

Under the covers, `Backbone.sync` is the function called every time Backbone tries to read or save models to the server. It uses jQuery or Zepto's ajax implementations to make these RESTful requests, however this can be overridden as per your needs.

The sync function may be overriden globally as Backbone.sync, or at a finer-grained level, by adding a sync function to a Backbone collection or to an individual model.

There's no fancy plugin API for adding a persistence layer – simply override Backbone.sync with the same function signature:

```
Backbone.sync = function(method, model, options) {
};
```

The default methodMap is useful for working out what the method argument does:

```
var methodMap = {
  'create': 'POST',
  'update': 'PUT',
  'delete': 'DELETE',
  'read':   'GET'
};
```

In the above example if we wanted to log an event when `.sync()` was called, we could do this:

```
var id_counter = 1;
Backbone.sync = function(method, model) {
  console.log("I\'ve been passed " + method + " with " + JSON.stringify(model));
  if(method === 'create'){ model.set('id', id_counter++); }
};
```

Resetting/Refreshing Collections

Rather than adding or removing models individually, you might occasionally wish to update an entire collection at once. `Collection.reset()` allows us to replace an entire collection with new models as follows:

```
var TodosCollection = new Backbone.Collection();

TodosCollection.on("reset", function() {
  console.log("Collection reseted.");
});

TodosCollection.add([
  { title: 'go to Jamaica.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Disneyland.', completed: true }
]);

console.log('Collection size: ' + TodosCollection.length);

TodosCollection.reset([
  { title: 'go to Cuba.', completed: false }
]);
console.log('Collection size: ' + TodosCollection.length);
```

Note that using `Collection.reset()` doesn't fire any `add` or `remove` events. A `reset` event is fired instead as shown in example.

Underscore utility functions

As Backbone requires Underscore as a hard dependency, we're able to use many of the utilities it has to offer to aid with our application development. Here's an example of how Underscore's `forEach` method that can be used for iterating over collection and `sortBy()` method that can be used to sort a collection of todos based on a particular attribute.

```
var TodosCollection = new Backbone.Collection();

TodosCollection.on("reset", function() {
  console.log("Collection reseted.");
});

TodosCollection.add([
  { title: 'go to Belgium.', completed: false },
  { title: 'go to China.', completed: false },
  { title: 'go to Austria.', completed: true }
]);

TodosCollection.forEach(function(model){
  console.log(model.get('title'));
});

var sortedByAlphabet = TodosCollection.sortBy(function (todo) {
    return todo.get("title").toLowerCase();
});

console.log("- Now sorted: ");

sortedByAlphabet.forEach(function(model){
  console.log(model.get('title'));
});
```

The complete list of what Underscore can do is beyond the scope of this guide, but can be found in its official docs.

Chainable API

Speaking of utility methods, another bit of sugar in Backbone is the support for Underscore's chain method. This works by calling the original method with the current array of models and returning the result. In case you haven't seen it before, the chainable API looks like this:

```
var collection = new Backbone.Collection([
  { name: 'Tim', age: 5 },
  { name: 'Ida', age: 26 },
  { name: 'Rob', age: 55 }
]);

var filteredNames = collection.chain()
  .filter(function(item) { return item.get('age') > 10; })
  .map(function(item) { return item.get('name'); })
  .value();

console.log(filteredNames); // logs: ['Ida', 'Rob']
```

Some of the Backbone-specific methods will return this, which means they can be chained as well:

```
var collection = new Backbone.Collection();

collection
    .add({ name: 'John', age: 23 })
    .add({ name: 'Harry', age: 33 })
    .add({ name: 'Steve', age: 41 });

var names = collection.pluck('name');

console.log(names); // logs: ['John', 'Harry', 'Steve']
```

Events

As we've covered, `Backbone.Events` is mixed into the other Backbone "classes", including:

- Backbone.Model
- Backbone.Collection
- Backbone.Router
- Backbone.History
- Backbone.View

Events are the standard way to deal with user interface actions, through the declarative event bindings on views, and also model and collection changes. Mastering events is one of the quickest ways to become more productive with Backbone.

`Backbone.Events` also has the ability to give any object a way to bind and trigger custom events. We can mix this module into any object easily and there isn't a requirement for events to be declared prior to them being bound.

Example:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);
```

```
// Add a custom event
ourObject.on('dance', function(msg){
  console.log('We triggered ' + msg);
});

// Trigger the custom event
ourObject.trigger('dance', 'our event');
```

If you're familiar with jQuery custom events or the concept of Publish/Subscribe, `Backbone.Events` provides a system that is very similar with `on` being analogous to `subscribe` and `trigger` being similar to `publish`.

`on` basically allows us to bind a callback function to any object, as we've done with `dance` in the above example. Whenever the event is fired, our callback is invoked.

The official Backbone.js documentation recommends namespacing event names using colons if you end up using quite a few of these on your page. e.g:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We started " + msg); }

// Add namespaced custom events
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);

// Trigger the custom events
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");

// This one triggers nothing as no listener listens for it
ourObject.trigger("dance", "break dancing. Yeah!");
```

A special `all` event is made available in case you would like an event to be triggered when any event occurs (e.g if you would like to screen events in a single location). The `all` event can be used as follows:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);
```

```
function dancing (msg) { console.log("We started " + msg); }

ourObject.on("all", function(eventName){
  console.log("The name of the event passed was " + eventName);
});

// This time each event will be catched with a catch 'all' event listener
ourObject.trigger("dance:tap", "tap dancing. Yeah!");
ourObject.trigger("dance:break", "break dancing. Yeah!");
ourObject.trigger("dance", "break dancing. Yeah!");
```

`off` allows us to remove a callback function that has previously been bound to an object. Going back to our Publish/Subscribe comparison, think of it as an `unsubscribe` for custom events.

To remove the `dance` event we previously bound to `ourObject`, we would simply do:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We  " + msg); }

// Add a namespaced custom events
ourObject.on("dance:tap", dancing);
ourObject.on("dance:break", dancing);

// Trigger the custom events. Each will be catched and acted upon.
ourObject.trigger("dance:tap", "started tap dancing. Yeah!");
ourObject.trigger("dance:break", "started break dancing. Yeah!");

// Removes event bound to the object
ourObject.off("dance:tap");

// Trigger the custom events again, but one is logged.
ourObject.trigger("dance:tap", "stopped tap dancing."); // won't be logged as it's not liste
ourObject.trigger("dance:break", "break dancing. Yeah!");
```

To remove all callbacks for the event we pass an event name (e.g `move`) to `off()` method on the object the event is bound to. If we wish to remove just a callback by a specific name, we can pass the callback name as second parameter:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function dancing (msg) { console.log("We are dancing. " + msg); }
function jumping (msg) { console.log("We are jumping. " + msg); }

// Add two listeners to the same event
ourObject.on("move", dancing);
ourObject.on("move", jumping);

// Trigger the events. Both listeners are called.
ourObject.trigger("move", "Yeah!");

// Removes specified listener
ourObject.off("move", dancing);

// Trigger the events again. One listener left.
ourObject.trigger("move", "Yeah, jump, jump!");
```

Finally, `trigger` triggers a callback for a specified event (or a space-separated list of events). e.g:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function doAction (msg) { console.log("We are " + msg); }

// Add event listeners
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
ourObject.on("skip", doAction);

// Single event
ourObject.trigger("dance", 'just dancing.');

// Multiple events
ourObject.trigger("dance jump skip", 'very tired from so much action.');
```

It is also possible to pass along additional arguments to each (or all) of these events via a second argument supported by `trigger`. e.g:

```
var ourObject = {};

// Mixin
_.extend(ourObject, Backbone.Events);

function doAction (actionObj) {
  console.log("We are " + actionObj.action + ' for ' + actionObj.duration );
}

// Add event listeners
ourObject.on("dance", doAction);
ourObject.on("jump", doAction);
ourObject.on("skip", doAction);

// Passing multiple arguments to single event
ourObject.trigger("dance", {duration: "5 minutes", action: 'dancing'});

// Passing multiple arguments to multiple events
ourObject.trigger("dance jump skip", {duration: "15 minutes", action: 'on fire'});
```

Routers

In Backbone, routers are used to help manage application state and for connecting URLs to application events. This is achieved using hash-tags with URL fragments, or using the browser's pushState and History API. Some examples of routes may be seen below:

```
http://example.com/#about
http://example.com/#search/seasonal-horns/page2
```

Note: An application will usually have at least one route mapping a URL route to a function that determines what happens when a user reaches that particular route. This relationship is defined as follows:

```
'route' : 'mappedFunction'
```

Let us now define our first router by extending `Backbone.Router`. For the purposes of this guide, we're going to continue pretending we're creating a complex todo application (something like personal organize/planner) that requires a complex TodoRouter.

Note the inline comments in the code example below as they continue the rest of the lesson on routers.

```
var TodoRouter = Backbone.Router.extend({
    /* define the route and function maps for this router */
    routes: {
        "about" : "showAbout",
        /* Sample usage: http://example.com/#about */

        "todo/:id" : "getTodo",
        /* This is an example of using a ":param" variable which allows us to match
        any of the components between two URL slashes */
        /* Sample usage: http://example.com/#todo/5 */

        "search/:query" : "searchTodos",
        /* We can also define multiple routes that are bound to the same map function,
        in this case searchTodos(). Note below how we're optionally passing in a
        reference to a page number if one is supplied */
        /* Sample usage: http://example.com/#search/job */

        "search/:query/p:page" : "searchTodos",
        /* As we can see, URLs may contain as many ":param"s as we wish */
        /* Sample usage: http://example.com/#search/job/p1 */

        "todos/:id/download/*documentPath" : "downloadDocument",
        /* This is an example of using a *splat. Splats are able to match any number of
        URL components and can be combined with ":param"s*/
        /* Sample usage: http://example.com/#todos/5/download/files/Meeting_schedule.doc */

        /* If you wish to use splats for anything beyond default routing, it's probably a g
        idea to leave them at the end of a URL otherwise you may need to apply regular
        expression parsing on your fragment */

        "*other"    : "defaultRoute"
        /* This is a default route that also uses a *splat. Consider the
        default route a wildcard for URLs that are either not matched or where
        the user has incorrectly typed in a route path manually */
        /* Sample usage: http://example.com/# <anything */
    },

    showAbout: function(){
    },

    getTodo: function(id){
        /*
        Note that the id matched in the above route will be passed to this function
        */
        console.log("You are trying to reach todo " + id);
    },
    searchTodos: function(query, page){
        var page_number = page || 1;
        console.log("Page number: " + page_number + " of the results for todos containing th
    },

    downloadDocument: function(id, path){
    },

    defaultRoute: function(other){
        console.log('Invalid. You attempted to reach:' + other);
    }
});

/* Now that we have a router setup, remember to instantiate it */

var myTodoRouter = new TodoRouter();
```

As of Backbone 0.5+, it's possible to opt-in for HTML5 pushState support via `window.history.pushState`. This permits you to define routes such as http://www.scriptjunkie.com/just/an/example. This will be supported with automatic degradation when a user's browser doesn't support pushState. For the purposes of this tutorial, we'll use the hashtag method.

Is there a limit to the number of routers I should be using? Andrew de Andrade has pointed out that DocumentCloud, the creators of Backbone, usually only use a single router in most of their applications. You're very likely to not require more than one or two routers in your own projects; the majority of your application routing can be kept organized in a single router without it getting unwieldy.

Backbone.history Next, we need to initialize `Backbone.history` as it handles `hashchange` events in our application. This will automatically handle routes that have been defined and trigger callbacks when they've been accessed.

The `Backbone.history.start()` method will simply tell Backbone that it's OK to begin monitoring all `hashchange` events as follows:

```
var TodoRouter = Backbone.Router.extend({
  /* define the route and function maps for this router */
  routes: {
    "about" : "showAbout",
    "search/:query" : "searchTodos",
    "search/:query/p:page" : "searchTodos"
  },

  showAbout: function(){},

  searchTodos: function(query, page){
    var page_number = page || 1;
    console.log("Page number: " + page_number + " of the results for todos containing the wo
  }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to and check console:
// http://localhost/#search/job/p3 logs: Page number: 3 of the results for todos containing
// http://localhost/#search/job logs: Page number: 1 of the results for todos containing the
// etc.
```

Note: To test the last example, you'll need to create a local development environment and test project, instructions for which are beyond the scope of what this book seeks to outline.

As an aside, if you would like to save application state to the URL at a particular point you can use the `.navigate()` method to achieve this. It simply updates your URL fragment without the need to trigger the `hashchange` event:

```
// Let's imagine we would like a specific fragment (edit) once a user opens a single todo
var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... other routes
  },

  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit'); // updates the fragment for us, but doesn't trigg
  },
  editTodo: function(id) {
    console.log("Edit todo openned.");
  }
});

var myTodoRouter = new TodoRouter();
```

```
Backbone.history.start();

// Go to:
// http://localhost/#todo/4 URL is updated to: http://localhost/#todo/4/edit
// but editTodo() function is not invoked even though location we end up is mapped to it.
//
// logs: View todo requested.
```

It is also possible for `Router.navigate()` to trigger the route as well as update the URL fragment.

```
var TodoRouter = Backbone.Router.extend({
  routes: {
    "todo/:id": "viewTodo",
    "todo/:id/edit": "editTodo"
    // ... other routes
  },

  viewTodo: function(id){
    console.log("View todo requested.");
    this.navigate("todo/" + id + '/edit', true); // updates the fragment and triggers the ro
  },
  editTodo: function(id) {
    console.log("Edit todo openned.");
  }
});

var myTodoRouter = new TodoRouter();

Backbone.history.start();

// Go to:
// http://localhost/#todo/4 url is updated to: http://localhost/#todo/4/edit
// but this time editTodo() function is invoked.
//
// logs:
// View todo requested.
// Edit todo openned.
```

Backbone's Sync API

The Backbone.sync method is intended to be overridden to support other backends. The built-in method is tailored to a certain breed of RESTful JSON APIs – Backbone was originally extracted from a Ruby on Rails application, which uses HTTP methods like PUT the same way.

This works via the model and collection classes' sync method, which calls Backbone.sync. Both will call this.sync internally when fetching, saving, or deleting items.

The sync method is called with three parameters:

- method: One of create, update, delete, read
- model: The Backbone model object
- options: May include success and error methods

Implementing a new sync method can use the following pattern:

```
Backbone.sync = function(method, model, options) {
  var requestContent = {}, success, error;

  function success(result) {
    // Handle results from MyAPI
    if (options.success) {
      options.success(result);
    }
  }

  function error(result) {
    // Handle results from MyAPI
    if (options.error) {
      options.error(result);
    }
  }

  options || (options = {});

  switch (method) {
    case 'create':
      requestContent['resource'] = model.toJSON();
      return MyAPI.create(model, success, error);

    case 'update':
      requestContent['resource'] = model.toJSON();
      return MyAPI.update(model, success, error);

    case 'delete':
      return MyAPI.destroy(model, success, error);

    case 'read':
      if (model.attributes[model.idAttribute]) {
        return MyAPI.find(model, success, error);
      } else {
        return MyAPI.findAll(model, success, error);
      }
  }
};
```

This pattern delegates API calls to a new object, which could be a Backbone-style class that supports events. This can be safely tested separately, and potentially used with libraries other than Backbone.

There are quite a few sync implementations out there:

- Backbone localStorage
- Backbone offline
- Backbone Redis
- backbone-parse
- backbone-websql
- Backbone Caching Sync

Conflict Management

Like most client-side projects, Backbone's code is wrapped in an immediately-invoked function expression:

```
(function(){
  // Backbone.js
}).call(this);
```

Several things happen during this configuration stage. A Backbone "namespace" is created, and multiple versions of Backbone on the same page are supported through the noConflict mode:

```
var root = this;
var previousBackbone = root.Backbone;

Backbone.noConflict = function() {
  root.Backbone = previousBackbone;
  return this;
};
```

Multiple versions of Backbone can be used on the same page by calling noConflict like this:

```
var Backbone19 = Backbone.noConflict();
// Backbone19 refers to the most recently loaded version,
// and `window.Backbone` will be restored to the previously
// loaded version
```

This initial configuration code also supports CommonJS modules so Backbone can be used in Node projects:

```
var Backbone;
if (typeof exports !== 'undefined') {
  Backbone = exports;
} else {
  Backbone = root.Backbone = {};
}
```

Inheritance & Mixins

For its inheritance, Backbone internally uses an `inherits` function inspired by `goog.inherits`, Google's implementation from the Closure Library. It's basically a function to correctly setup the prototype chain.

```
 var inherits = function(parent, protoProps, staticProps) {
      ...
```

The only major difference here is that Backbone's API accepts two objects containing "instance" and "static" methods.

Following on from this, for inheritance purposes all of Backbone's objects contain an `extend` method as follows:

```
Model.extend = Collection.extend = Router.extend = View.extend = extend;
```

Most development with Backbone is based around inheriting from these objects, and they're designed to mimic a classical object-oriented implementation.

If this sounds familiar, it's because `extend` is an Underscore.js utility, although Backbone itself does a lot more with this. See below for Underscore's `extend`:

```
each(slice.call(arguments, 1), function(source) {
  for (var prop in source) {
    obj[prop] = source[prop];
  }
});
return obj;
```

The above isn't quite the same as ES5's `Object.create`, as it's actually copying properties (methods and values) from one object to another. As this isn't enough to support Backbone's inheritance and class model, the following steps are performed:

- The instance methods are checked to see if there's a constructor property. If so, the class's constructor is used, otherwise the parent's constructor is used (i.e., Backbone.Model)
- Underscore's extend method is called to add the parent class's methods to the new child class
- The `prototype` property of a blank constructor function is assigned with the parent's prototype, and a new instance of this is set to the child's `prototype` property
- Underscore's extend method is called twice to add the static and instance methods to the child class
- The child's prototype's constructor and a `__super__` property are assigned
- This pattern is also used for classes in CoffeeScript, so Backbone classes are compatible with CoffeeScript classes.

`extend` can be used for a great deal more and developers who are fans of mixins will like that it can be used for this too. You can define functionality on any custom object, and then quite literally copy & paste all of the methods and attributes from that object to a Backbone one:

For example:

```
var MyMixin = {
  foo: 'bar',
  sayFoo: function(){alert(this.foo);}
};

var MyView = Backbone.View.extend({
 // ...
});

_.extend(MyView.prototype, MyMixin);

var myView = new MyView();
myView.sayFoo(); //=> 'bar'
```

We can take this further and also apply it to View inheritance. The following is an example of how to extend one View using another:

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

```
var PanelAdvanced = Panel.extend({
});
```

However, if you have an `initialize()` method in Panel, then it won't be called if you also have an `initialize()` method in PanelAdvanced, so you would have to call Panel's initialize method explicitly:

```
var Panel = Backbone.View.extend({
  initialize: function(options){
    console.log('Panel initialized');
    this.foo = 'bar';
  }
});

var PanelAdvanced = Panel.extend({
  initialize: function(options){
    Panel.prototype.initialize.call(this, [options]);
    console.log('PanelAdvanced initialized');
    console.log(this.foo); // Log: bar
  }
});

// We can also inherit PanelAdvaned if needed
var PanelAdvancedExtra = PanelAdvanced.extend({
  initialize: function(options){
    PanelAdvanced.prototype.initialize.call(this, [options]);
    console.log('PanelAdvancedExtra initialized');
  }
});

new Panel();
new PanelAdvanced();
new PanelAdvancedExtra();
```

This isn't the most elegant of solutions because if you have a lot of Views that inherit from Panel, then you'll have to remember to call Panel's initialize from all of them.

It's worth noting that if Panel doesn't have an initialize method now but you choose to add it in the future, then you'll need to go to all of the inherited classes in the future and make sure they call Panel's initialize.

So here's an alternative way to define Panel so that your inherited views don't need to call Panel's initialize method:

```
var Panel = function (options) {
  // put all of Panel's initialization code here
  console.log('Panel initialized');
  this.foo = 'bar';

  Backbone.View.apply(this, [options]);
};

_.extend(Panel.prototype, Backbone.View.prototype, {
  // put all of Panel's methods here. For example:
  sayHi: function () {
    console.log('hello from Panel');
  }
});

Panel.extend = Backbone.View.extend;

// other classes then inherit from Panel like this:
var PanelAdvanced = Panel.extend({
  initialize: function (options) {
    console.log('PanelAdvanced initialized');
    console.log(this.foo);
  }
});

var panelAdvanced = new PanelAdvanced(); //Logs: Panel initialized, PanelAdvanced initializ
panelAdvanced.sayHi(); // Logs: hello from Panel
```

When used appropriately, Underscore's `extend` method can save a great deal of time and effort writing redundant code.

(Thanks to Alex Young, Derick Bailey and JohnnyO for the heads up about these tips).

Backbone-Super Backbone-Super by Lukas Olson adds a _*super* method to *Backbone.Model* using John Resig's Inheritance script. Rather than using Backbone.Model.prototype.set.call as per the Backbone.js documentation, _super can be called instead:

```
// This is how we normally do it
var OldFashionedNote = Backbone.Model.extend({
  set: function(attributes, options) {
    // Call parent's method
    Backbone.Model.prototype.set.call(this, attributes, options);
    // some custom code here
    // ...
  }
});
```

After including this plugin, you can do the same thing with the following syntax:

```
// This is how we can do it after using the Backbone-super plugin
var Note = Backbone.Model.extend({
  set: function(attributes, options) {
    // Call parent's method
    this._super(attributes, options);
    // some custom code here
    // ...
  }
});
```

Dependencies

The official Backbone.js documentation states:

> Backbone's only hard dependency is Underscore.js ($>$ 1.3.1). For RESTful persistence, history support via Backbone.Router and DOM manipulation with Backbone.View, include json2.js, and either jQuery ($>$ 1.4.2) or Zepto.

Lo-dash, a fork of Underscore containing performance enhancements is also compatible with Backbone.

What this translates to is that if you require working with anything beyond models, you will need to include a DOM manipulation library such as jQuery or Zepto. Underscore is primarily used for it's utility methods (which Backbone relies upon heaviy) and json2.js for legacy browser JSON support if Backbone.sync is used.

DOM Manipulation

Although most developers won't need it, Backbone does support setting a custom DOM library to be used instead of these options. From the source:

```
// Set the JavaScript library that will be used for DOM manipulation and
// Ajax calls (a.k.a. the `$` variable). By default Backbone will use: jQuery,
// Zepto, or Ender; but the `setDomLibrary()` method lets you inject an
// alternate JavaScript library (or a mock library for testing your views
// outside of a browser).

Backbone.setDomLibrary = function(lib) {
  $ = lib;
};
```

Calling this method will allow you to use any custom DOM-manipulation library. e.g:

```
Backbone.setDomLibrary(aCustomLibrary);
```

Utilities

Underscore.js is heavily used in Backbone behind the scenes for everything from object extension through to event binding. As the entire library is generally included, we get free access to a number of useful utilities we can use on Collections such as filtering `_.filter()`, sorting `_.sortBy()`, mapping `_.map()` and so on.

From the source:

```
// Underscore methods that we want to implement on the Collection.
var methods = ['forEach', 'each', 'map', 'reduce', 'reduceRight', 'find',
    'detect', 'filter', 'select', 'reject', 'every', 'all', 'some', 'any',
    'include', 'contains', 'invoke', 'max', 'min', 'sortBy', 'sortedIndex',
    'toArray', 'size', 'first', 'initial', 'rest', 'last', 'without', 'indexOf',
    'shuffle', 'lastIndexOf', 'isEmpty', 'groupBy'];

// Mix in each Underscore method as a proxy to Collection#models.
  _.each(methods, function(method) {
    Collection.prototype[method] = function() {
      return _[method].apply(_, [this.models].concat(_.toArray(arguments)));
    };
```

However, for a complete linked list of methods supported, see the official documentation.

RESTFul persistence

Models and collections in Backbone can be "sync"ed with the server using the `fetch`, `save` and `destroy` methods. All of these methods delegate back to the `Backbone.sync` function, which actually wraps jQuery/Zepto's `$.ajax` function, calling GET, POST and DELETE for the respective persistence methods on Backbone models.

From the the source for `Backbone.sync`:

```
var methodMap = {
    'create': 'POST',
    'update': 'PUT',
    'delete': 'DELETE',
    'read':   'GET'
  };

Backbone.sync = function(method, model, options) {
    var type = methodMap[method];
    options || (options = {});
    // ... Followed by lots of Backbone.js configuration, then..
   return $.ajax(_.extend(params, options));
```

Routing

Calls to `Backbone.History.start` rely on jQuery/Zepto binding `popState` or `hashchange` event listeners back to the window object.

From the source for `Backbone.history.start`:

```
// Depending on whether we're using pushState or hashes, and whether 'onhashchange' is
// supported, determine how we check the URL state.
if (this._hasPushState) {
        $(window).bind('popstate', this.checkUrl);
      } else if (this._wantsHashChange && ('onhashchange' in window) && !oldIE) {
        $(window).bind('hashchange', this.checkUrl);
      ...
```

`Backbone.History.stop` similarly uses your DOM manipulation library to unbind these event listeners.

Namespacing

When learning how to use Backbone, an important and commonly overlooked area by tutorials is namespacing. If you already have experience with namespacing in JavaScript, the following section will provide some advice on how to specifically apply concepts you know to Backbone, however I will also be covering explanations for beginners to ensure everyone is on the same page.

What is namespacing? The basic idea around namespacing is to avoid collisions with other objects or variables in the global namespace. They're important as it's best to safeguard your code from breaking in the event of another script on the page using the same variable names as you are. As a good 'citizen' of the global namespace, it's also imperative that you do your best to similarly not prevent other developer's scripts executing due to the same issues.

JavaScript doesn't really have built-in support for namespaces like other languages, however it does have closures which can be used to achieve a similar effect.

In this section we'll be taking a look shortly at some examples of how you can namespace your models, views, routers and other components specifically. The patterns we'll be examining are:

- Single global variables
- Object Literals
- Nested namespacing

Single global variables

One popular pattern for namespacing in JavaScript is opting for a single global variable as your primary object of reference. A skeleton implementation of this where we return an object with functions and properties can be found below:

```
var myApplication = (function(){
    function(){
      // ...
    },
    return {
      // ...
    }
})();
```

You've probably seen this technique before. A Backbone-specific example might look like this:

```
var myViews = (function(){
    return {
        TodoView: Backbone.View.extend({ .. }),
        TodosView: Backbone.View.extend({ .. }),
        AboutView: Backbone.View.extend({ .. });
        //etc.
    };
})();
```

Here we can return a set of views, but the same technique could return an entire collection of models, views and routers depending on how you decide to structure your application. Although this works for certain situations, the biggest challenge with the single global variable pattern is ensuring that no one else has used the same global variable name as you have in the page.

One solution to this problem, as mentioned by Peter Michaux, is to use prefix namespacing. It's a simple concept at heart, but the idea is you select a common prefix name (in this example, `myApplication_`) and then define any methods, variables or other objects after the prefix.

```
var myApplication_todoView = Backbone.View.extend({}),
    myApplication_todosView = Backbone.View.extend({});
```

This is effective from the perspective of trying to lower the chances of a particular variable existing in the global scope, but remember that a uniquely named object can have the same effect. This aside, the biggest issue with the pattern is that it can result in a large number of global objects once your application starts to grow.

For more on Peter's views about the single global variable pattern, read his excellent post on them.

Note: There are several other variations on the single global variable pattern out in the wild, however having reviewed quite a few, I felt the prefixing approach applied best to Backbone.

Object Literals

Object Literals have the advantage of not polluting the global namespace but assist in organizing code and parameters logically. They're beneficial if you wish to create easily readable structures that can be expanded to support deep nesting. Unlike simple global variables, Object Literals often also take into account tests for the existence of a variable by the same name, which helps reduce the chances of collision.

This example demonstrates two ways you can check to see if a namespace already exists before defining it. I commonly use Option 2.

```
/* Doesn't check for existence of myApplication */
var myApplication = {};

/*
Does check for existence. If already defined, we use that instance.
Option 1:   if(!myApplication) myApplication = {};
Option 2:   var myApplication = myApplication || {};
We can then populate our object literal to support models, views and collections (or any da
*/

var myApplication = {
    models : {},
    views : {
        pages : {}
    },
    collections : {}
};
```

One can also opt for adding properties directly to the namespace (such as your views, in the following example):

```
var myTodosViews = myTodosViews || {};
myTodosViews.todoView = Backbone.View.extend({});
myTodosViews.todosView = Backbone.View.extend({});
```

The benefit of this pattern is that you're able to easily encapsulate all of your models, views, routers etc. in a way that clearly separates them and provides a solid foundation for extending your code.

This pattern has a number of benefits. It's often a good idea to decouple the default configuration for your application into a single area that can be easily modified without the need to search through your entire codebase just to alter it. Here's an example of a hypothetical object literal that stores application configuration settings:

```
var myConfig = {
  language: 'english',
  defaults: {
    enableDelegation: true,
    maxTodos: 40
  },
  theme: {
    skin: 'a',
    toolbars: {
      index: 'ui-navigation-toolbar',
      pages: 'ui-custom-toolbar'
    }
  }
}
```

Note that there are really only minor syntactical differences between the Object Literal pattern and a standard JSON data set. If for any reason you wish to use JSON for storing your configurations instead (e.g. for simpler storage when sending to the back-end), feel free to.

For more on the Object Literal pattern, I recommend reading Rebecca Murphey's excellent article on the topic.

Nested namespacing

An extension of the Object Literal pattern is nested namespacing. It's another common pattern used that offers a lower risk of collision due to the fact that even if a top-level namespace already exists, it's unlikely the same nested children do. For example, Yahoo's YUI uses the nested object namespacing pattern extensively:

```
YAHOO.util.Dom.getElementsByClassName('test');
```

Yahoo's YUI uses the nested object namespacing pattern regularly and even DocumentCloud (the creators of Backbone) use the nested namespacing pattern in their main applications. A sample implementation of nested namespacing with Backbone may look like this:

```
var todoApp =  todoApp || {};

// perform similar check for nested children
todoApp.routers = todoApp.routers || {};
todoApp.model = todoApp.model || {};
todoApp.model.special = todoApp.model.special || {};

// routers
todoApp.routers.Workspace   = Backbone.Router.extend({});
todoApp.routers.TodoSearch = Backbone.Router.extend({});

// models
todoApp.model.Todo   = Backbone.Model.extend({});
todoApp.model.Notes = Backbone.Model.extend({});

// special models
todoApp.model.special.Admin = Backbone.Model.extend({});
```

This is readable, clearly organized, and is a relatively safe way of namespacing your Backbone application. The only real caveat however is that it requires your browser's JavaScript engine to first locate the todoApp object, then dig down until it gets to the function you're calling. However, developers such as Juriy Zaytsev (kangax) have tested and found the performance differences between single object namespacing vs the 'nested' approach to be quite negligible.

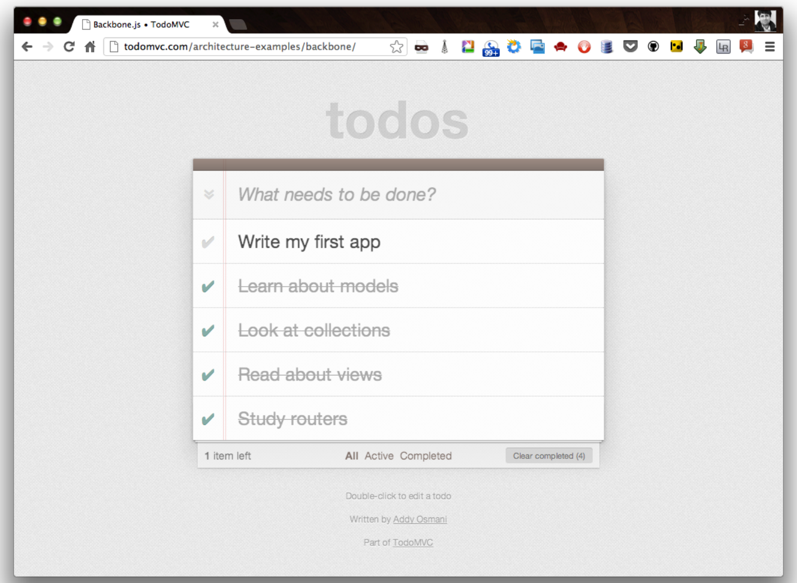
Recommendation

Reviewing the namespace patterns above, the option that I prefer when writing Backbone applications is nested object namespacing with the object literal pattern.

Single global variables may work fine for applications that are relatively trivial. However, larger codebases requiring both namespaces and deep sub-namespaces require a succinct solution that's both readable and scalable. I feel this pattern achieves both of these objectives and is a good choice for most Backbone development.

Exercise 1: Todos - Your First Backbone.js App

Now that we've covered fundamentals, let's write our first Backbone.js application. We'll build the Backbone Todo List application exhibited on TodoMVC.com. Building a Todo List is a great way to learn Backbone's conventions. It's a relatively simple application, yet technical challenges surrounding binding, persisting model data, routing and template rendering provide the opportunity to illustrate some core Backbone features.



Let's consider the application's architecture at a high level. We'll need:

- a `Todo` model to describe individual todo items
- a `TodoList` collection to store and persist todos
- a way of creating todos
- a way to display a listing of todos
- a way to edit existing todos

- a way to deem a todo complete
- a way to delete todos
- a way to bookmark the items that have been completed or are remaining

Essentially, these features are classic CRUD methods. Let's get started!

Index

First, we'll set up the basic application dependencies: jQuery, Underscore, Backbone.js and the Backbone LocalStorage adapter. These will be loaded in index.html, the application's sole HTML file:

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <title>Backbone.js • TodoMVC</title>
  <link rel="stylesheet" href="assets/base.css">
</head>
<body>
  <script src="js/lib/jquery.min.js"></script>
  <script src="js/lib/underscore-min.js"></script>
  <script src="js/lib/backbone-min.js"></script>
  <script src="js/lib/backbone.localStorage.js"></script>
  <script src="js/models/todo.js"></script>
  <script type="text/template" id="item-template"></script>
  <script type="text/template" id="stats-template"></script>
  <script src="js/collections/todos.js"></script>
  <script src="js/views/todo.js"></script>
  <script src="js/views/app.js"></script>
  <script src="js/routers/router.js"></script>
  <script src="js/app.js"></script>
</body>
</html>
```

In addition to the aforementioned application dependencies, note that a few other, application-specific files are also loaded. These are organized into folders representing their application responsibilities: models, views, collections and routers. An app.js file is present to house central initialization code.

Note: If you want to follow along, create a directory structure as demonstrated in index.html. You will also need base.css and bg.png, which can both live in an assets directory. And again, the final application can be demo'd at TodoMVC.com.

Application HTML

Now let's take a look at our application's static HTML. We'll need an `<input>` for creating new todos, a `<ul id="todo-list" />` for listing the actual todos and a section containing some operations, such as clearing completed todos.

```
<section id="todoapp">
  <header id="header">
    <h1>todos</h1>
    <input id="new-todo" placeholder="What needs to be done?" autofocus>
  </header>
  <section id="main">
    <input id="toggle-all" type="checkbox">
    <label for="toggle-all">Mark all as complete</label>
    <ul id="todo-list"></ul>
  </section>
  <footer id="footer"></footer>
</section>
<div id="info">
  <p>Double-click to edit a todo</p>
  <p>Written by <a href="https://github.com/addyosmani">Addy Osmani</a></p>
  <p>Part of <a href="http://todomvc.com">TodoMVC</a></p>
</div>
```

We'll populate our todo-list and add a statistics section containing details about what incomplete items later on.

In order to implement these features, we'll return to the fundamentals: a Todo model.

Todo model

The `Todo` model is remarkably straightforward. First, a todo has two attributes: a `title` stores todo item's title and a `completed` status indicates if it's complete. These attributes are passed as defaults, as is exemplified below:

```
// js/models/todo.js

var app = app || {};

// Todo Model
// ----------
// Our basic **Todo** model has `title` and `completed` attributes.
```

```
app.Todo = Backbone.Model.extend({

  // Default attributes for the todo
  // and ensure that each todo created has `title` and `completed` keys.
  defaults: {
    title: '',
    completed: false
  },

  // Toggle the `completed` state of this todo item.
  toggle: function() {
    this.save({
      completed: !this.get('completed')
    });
  }

});
```

The Todo model also features a `toggle()` method through which a Todo item's completion status can be set.

Todo collection

Next, a `TodoList` collection is used to group our models. The collection is extended by localStorage, which automatically persists Todo records to HTML5 Local Storage via the Backbone LocalStorage adapter. Through local storage, they're saved between page requests.

The collection's `completed()` and `remaining()` methods return an array of unfinished and finished todos, respectively.

A `nextOrder()` method keeps our Todo items in sequential order while a `comparator()` sorts items by their insertion order.

```
// js/collections/todos.js

var app = app || {};

// Todo Collection
// ---------------

// The collection of todos is backed by *localStorage* instead of a remote
// server.
var TodoList = Backbone.Collection.extend({
  // Reference to this collection's model.
  model: app.Todo,

  // Save all of the todo items under the `"todos"` namespace.
  localStorage: new Backbone.LocalStorage('todos-backbone'),

  // Filter down the list of all todo items that are finished.
  completed: function() {
    return this.filter(function( todo ) {
      return todo.get('completed');
    });
  },

  // Filter down the list to only todo items that are still not finished.
  remaining: function() {
    return this.without.apply( this, this.completed() );
  },

  // We keep the Todos in sequential order, despite being saved by unordered
  // GUID in the database. This generates the next order number for new items.
  nextOrder: function() {
    if ( !this.length ) {
      return 1;
    }
    return this.last().get('order') + 1;
  },

  // Todos are sorted by their original insertion order.
  comparator: function( todo ) {
    return todo.get('order');
  }
});

// Create our global collection of **Todos**.
app.Todos = new TodoList();
```

Application View

Let's examine the core of the application's logic, the views. Each view supports functionality such as edit-in-place, and is therefore associated with a fair amount of logic. To help organize this logic, we'll utilize the element controller pattern. The element controller pattern consists of two views: one controls a collection of items while the other deals with each individual item.

In other words, one view, `AppView`, will handle the creation of new todos, as well as rendering the initial todo list. Instances of another view, `TodoView`, will be associated with an individual Todo record. Todo instances can handle editing, updating and destroying their associated todo.

To keep things simple, the application will be 'read-only', at least for now. At this stage, it won't support functionality for creating, editing or deleting todos:

```
// js/views/app.js

var app = app || {};

// The Application
// ---------------

// Our overall **AppView** is the top-level piece of UI.
app.AppView = Backbone.View.extend({

  // Instead of generating a new element, bind to the existing skeleton of
  // the App already present in the HTML.
  el: '#todoapp',

  // Our template for the line of statistics at the bottom of the app.
  statsTemplate: _.template( $('#stats-template').html() ),

  // At initialization we bind to the relevant events on the `Todos`
  // collection, when items are added or changed. Kick things off by
  // loading any preexisting todos that might be saved in *localStorage*.
  initialize: function() {
    this.input = this.$('#new-todo');
    this.allCheckbox = this.$('#toggle-all')[0];
    this.$footer = this.$('#footer');
    this.$main = this.$('#main');

    window.app.Todos.on( 'add', this.addOne, this );
    window.app.Todos.on( 'reset', this.addAll, this );
    window.app.Todos.on( 'all', this.render, this );

    app.Todos.fetch();
  },

  // Re-rendering the App just means refreshing the statistics -- the rest
  // of the app doesn't change.
  render: function() {
    var completed = app.Todos.completed().length;
```

```
      var remaining = app.Todos.remaining().length;

      if ( app.Todos.length ) {
        this.$main.show();
        this.$footer.show();

        this.$footer.html(this.statsTemplate({
          completed: completed,
          remaining: remaining
        }));

      } else {
        this.$main.hide();
        this.$footer.hide();
      }

      this.allCheckbox.checked = !remaining;
    },

    // Add a single todo item to the list by creating a view for it, and
    // appending its element to the `<ul>`.
    addOne: function( todo ) {
      var view = new app.TodoView({ model: todo });
      $('#todo-list').append( view.render().el );
    },

    // Add all items in the **Todos** collection at once.
    addAll: function() {
      this.$('#todo-list').html('');
      app.Todos.each(this.addOne, this);
    }

  });
```

A few notable features are present in the AppView, including a `statsTemplate` method, an `initialize` method that's called on instantiation, and several view-specific methods.

An `el` (element) property stores a selector targeting the DOM element with an ID of `todoapp`. In the case of our application, `el` refers to the matching `<section id="todoapp" />` element in index.html.

Let's take a look at the constructor function. It's binding to several events on the Todo model, such as `add`, `reset` and `all`. Since we're delegating handling of updates and deletes to the `TodoView` view, we don't need to worry about that here. The two pieces of logic are:

- When a new todo is created, the `add` event will be fired, calling `addOne()`, which instantiates the TodoView view, rendering it and appending the resultant element to our Todo list.
- When a `reset` event is called (i.e. we wish to update the collection in bulk such as when the Todos have been loaded from Local Storage), `addAll()` is called, which iterates over all of the Todos currently in our collection and fires `addOne()` for each item.

We can then add in the logic for creating new todos, editing them and filtering them based on their completed status.

- events: We define an events hash containing declarative callbacks for our DOM events.
- `createOnEnter()`: Creates a new Todo model which persists in localStorage when a user hits return inside the `<input/>` field and resets the main `<input/>` field value to prepare it for the next entry. This creates the model via newAttributes(), which is an object literal composed of the title, order and completed state of the new item being added.
- `clearCompleted()`: Removes the items in the todo list that have been marked as completed.
- `toggleAllComplete()`: Allows a user to set all of the items in the todo list to completed.
- `initialize()`: We bind a callback for a change:completed event, letting us know a change has been made as well to an existing todo item. We also bind a callback for a filter event, which works a little similar to addOne() and addAll(). It's responsibility is to toggle what todo items are visible based on the filter currently selected in the UI (all, completed or remaining) through filterOne() and filterAll().
- `render()`: We add some conditional CSS styling based on the filter currently selected so that the selected route is highlighted.

```
// js/views/app.js

var app = app || {};

// The Application
// ---------------

// Our overall **AppView** is the top-level piece of UI.
app.AppView = Backbone.View.extend({

  // Instead of generating a new element, bind to the existing skeleton of
  // the App already present in the HTML.
```

```
el: '#todoapp',

// Our template for the line of statistics at the bottom of the app.
statsTemplate: _.template( $('#stats-template').html() ),

// Delegated events for creating new items, and clearing completed ones.
events: {
  'keypress #new-todo': 'createOnEnter',
  'click #clear-completed': 'clearCompleted',
  'click #toggle-all': 'toggleAllComplete'
},

// At initialization we bind to the relevant events on the `Todos`
// collection, when items are added or changed. Kick things off by
// loading any preexisting todos that might be saved in *localStorage*.
initialize: function() {
  this.input = this.$('#new-todo');
  this.allCheckbox = this.$('#toggle-all')[0];
  this.$footer = this.$('#footer');
  this.$main = this.$('#main');

  window.app.Todos.on( 'add', this.addOne, this );
  window.app.Todos.on( 'reset', this.addAll, this );
  window.app.Todos.on( 'change:completed', this.filterOne, this );
  window.app.Todos.on( 'filter', this.filterAll, this );

  window.app.Todos.on( 'all', this.render, this );

  app.Todos.fetch();
},

// Re-rendering the App just means refreshing the statistics -- the rest
// of the app doesn't change.
render: function() {
  var completed = app.Todos.completed().length;
  var remaining = app.Todos.remaining().length;

  if ( app.Todos.length ) {
    this.$main.show();
    this.$footer.show();

    this.$footer.html(this.statsTemplate({
      completed: completed,
      remaining: remaining
    }));
    this.$('#filters li a')
      .removeClass('selected')
      .filter('[href="#/' + ( app.TodoFilter || '' ) + '"]')
      .addClass('selected');
  } else {
    this.$main.hide();
    this.$footer.hide();
  }

  this.allCheckbox.checked = !remaining;
},

// Add a single todo item to the list by creating a view for it, and
// appending its element to the `<ul>`.
addOne: function( todo ) {
  var view = new app.TodoView({ model: todo });
  $('#todo-list').append( view.render().el );
},

// Add all items in the **Todos** collection at once.
addAll: function() {
  this.$('#todo-list').html('');
  app.Todos.each(this.addOne, this);
},

filterOne : function (todo) {
  todo.trigger('visible');
},

filterAll : function () {
  app.Todos.each(this.filterOne, this);
},

// Generate the attributes for a new Todo item.
newAttributes: function() {
  return {
    title: this.input.val().trim(),
    order: app.Todos.nextOrder(),
    completed: false
  };
},

// If you hit return in the main input field, create new **Todo** model,
// persisting it to *localStorage*.
createOnEnter: function( e ) {
  if ( e.which !== ENTER_KEY || !this.input.val().trim() ) {
        return;
      }

      app.Todos.create( this.newAttributes() );
      this.input.val('');
    },

    // Clear all completed todo items, destroying their models.
    clearCompleted: function() {
      _.each( window.app.Todos.completed(), function( todo ) {
        todo.destroy();
      });

      return false;
    },

    toggleAllComplete: function() {
      var completed = this.allCheckbox.checked;

      app.Todos.each(function( todo ) {
        todo.save({
          'completed': completed
        });
      });
    }
  });
```

Individual Todo View

Let's look at the `TodoView` view, now. This will be in charge of individual Todo records, making sure the view updates when the todo does. To enable this functionality, we should add event listeners to the view that will listen to the events on an individual todo's HTML representation.

```
  // js/views/todo.js

  var app = app || {};

  // Todo Item View
  // --------------

  // The DOM element for a todo item...
  app.TodoView = Backbone.View.extend({
```

```
//... is a list tag.
tagName:  'li',

// Cache the template function for a single item.
template: _.template( $('#item-template').html() ),

// The DOM events specific to an item.
events: {
  'dblclick label': 'edit',
  'keypress .edit': 'updateOnEnter',
  'blur .edit': 'close'
},

// The TodoView listens for changes to its model, re-rendering. Since there's
// a one-to-one correspondence between a **Todo** and a **TodoView** in this
// app, we set a direct reference on the model for convenience.
initialize: function() {
  this.model.on( 'change', this.render, this );
},

// Re-renders the todo item to the current state of the model and
// updates the reference to the todo's edit input within the view.
render: function() {
  this.$el.html( this.template( this.model.toJSON() ) );
  this.input = this.$('.edit');
  return this;
},

// Switch this view into `"editing"` mode, displaying the input field.
edit: function() {
  this.$el.addClass('editing');
  this.input.focus();
},

// Close the `"editing"` mode, saving changes to the todo.
close: function() {
  var value = this.input.val().trim();

  if ( value ) {
    this.model.save({ title: value });
  }

  this.$el.removeClass('editing');
},

// If you hit `enter`, we're through editing the item.
```

```
    updateOnEnter: function( e ) {
      if ( e.which === ENTER_KEY ) {
        this.close();
      }
    }
  });
```

In the `initialize()` constructor, we set up a listener tthat monitors a todo model's `change` event. In other words, when the todo updates, the application should re-render the view and visually reflect its changes.

In the `render()` method, we render an Underscore.js JavaScript template, called `#item-template`, which was previously compiled into this.template using Underscore's `_.template()` method. This returns an HTML fragment that replaces the view's current element. In other words, the rendered template is now present under `this.el` and can be appended to the todo list in the user interface.

Our events hash includes three callbacks:

- `edit()`: changes the current view into editing mode when a user double-clicks on an existing item in the todo list. This allows them to change the existing value of the item's title attribute.
- `updateOnEnter()`: checks that the user has hit the return/enter key and executes the close() function.
- `close()`: trims the value of the current text in our `<input/>` field, ensuring that we don't process it further if it contains no text (e.g ''). If a valid value has been provided, we save the changes to the current todo model and close editing mode by removing the corresponding CSS class.

Setup

So now we have two views: `AppView` and `TodoView`. The former needs to be instantiated on page load so its code is executed. This can be accomplished through jQuery's `ready()` utility, which will execute a function when the DOM is loaded.

```
  // js/app.js

  var app = app || {};
  var ENTER_KEY = 13;

  $(function() {
    // Kick things off by creating the **App**.
    new app.AppView();

  });
```

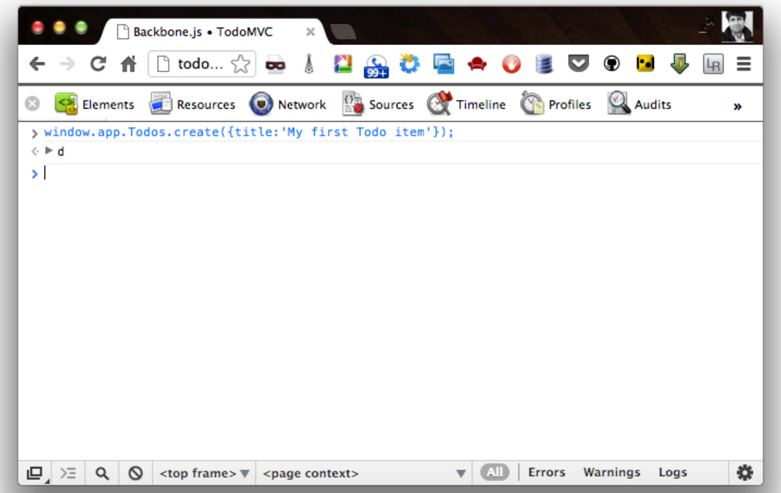
In action

Let's pause and ensure that the work we've done functions as intended.

If you are following along, open index.html in your web browser and monitor the console. If all is well, you shouldn't see any JavaScript errors. The todo list should be blank as we haven't yet created any todos. Plus, there is some additional work we'll need to do before the user interface fully functions.

However, a few things can be tested through the JavaScript console.

In the console, add a new todo item: `window.app.Todos.create({ title: 'My first Todo item'});` and hit return.



```
> window.app.Todos.create({title:'My first Todo item'});
<· ▶ d
> |
```

If all is functioning properly, this should log the new todo we've just added to the todos collection. The newly created todo is also saved to Local Storage and will be available on page refresh.

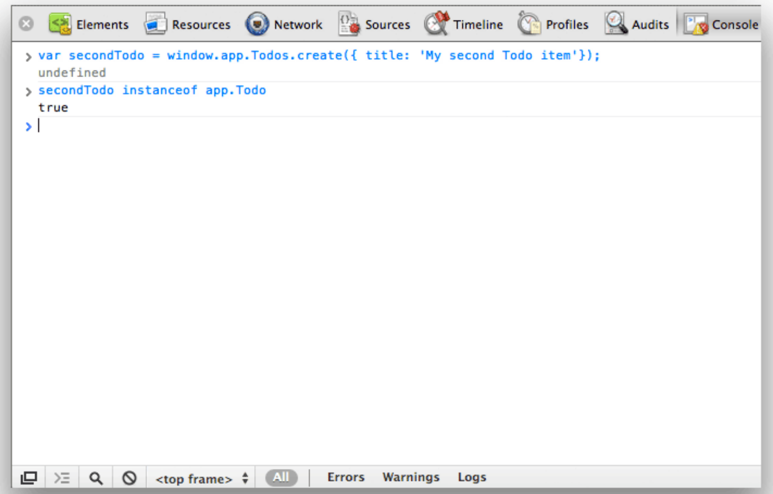
`window.app.Todos.create()` executes a collection method (`collection.create(attributes, [options])`) which instantiates a new model item of the type passed into the collection definition, in our case `app.Todo`:

```
var TodoList = Backbone.Collection.extend({

    model: app.Todo // the model type used by collection.create() to instantiate new mode
    ...
});
```

Run this into console to check it out:

`var secondTodo = window.app.Todos.create({ title: 'My second Todo item'});`

`secondTodo instanceof app.Todo`



```
> var secondTodo = window.app.Todos.create({ title: 'My second Todo item'});
  undefined
> secondTodo instanceof app.Todo
  true
>
```

Templates

The `#item-template` used in the `TodoView` view needs defining, so let's do that. One way of including templates in the page is by using custom script tags. These don't get evaluated by the browser, which just interprets them as plain text. Underscore micro-templating can then access the templates, rendering pieces of HTML.

```
<!-- index.html -->

<script type="text/template" id="item-template">
```

```
    <div class="view">
      <input class="toggle" type="checkbox" <%= completed ? 'checked' : '' %>>
      <label><%= title %></label>
      <button class="destroy"></button>
    </div>
    <input class="edit" value="<%= title %>">
  </script>
```

The template tags demonstrated above, such as `<%=` , are specific to Underscore.js, and documented on the Underscore site. In your own applications, you have a choice of template libraries, such as Mustache or Handlebars. Use whichever you prefer, Backbone doesn't mind.

Now when `_.template( $('#item-template').html() )` is called in the `TodoView` view our template will render correctly.

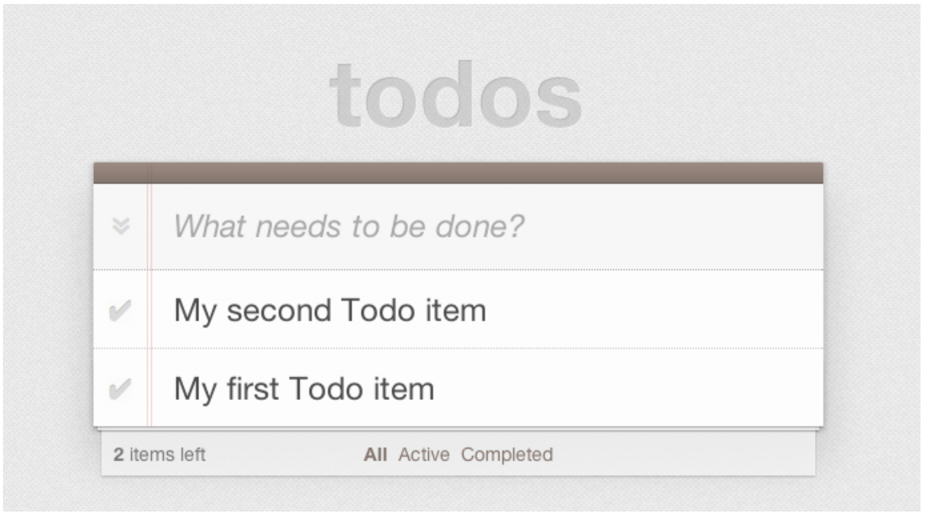
We also need to define #stats-template template we use to display how many items have been completed, as well as allowing the user to clear these items.

```
  <!-- index.html -->

  <script type="text/template" id="stats-template">
    <span id="todo-count"><strong><%= remaining %></strong> <%= remaining === 1 ? 'item' : '
    <ul id="filters">
      <li>
        <a class="selected" href="#/">All</a>
      </li>
      <li>
        <a href="#/active">Active</a>
      </li>
      <li>
        <a href="#/completed">Completed</a>
      </li>
    </ul>
    <% if (completed) { %>
    <button id="clear-completed">Clear completed (<%= completed %>)</button>
    <% } %>
  </script>
```

In action

Now refresh index.html and we should be able to see the fruits of our labour.

The todos added through console earlier should appear in the list populated from the Local Storage. Also, we should be able to type a todo name, and press return to submit the form, creating a new todo.



Excellent, we're making great progress, but how about completing and deleting todos?

Completing & deleting todos

So the next part of our tutorial is going to cover completing and deleting todos. These two actions are specific to each Todo item, so we need to add this functionality to the TodoView view.

The key part of this is the two event handlers we've added, a togglecompleted event on the todo's checkbox, and a click event on the todo's `<button class="destroy" />` button.

The checkbox's togglecompleted event invokes the toggle() function, which toggles the todos's completed status, then resaving the todo - very straightforward! The button's click event invokes `clear()`, which will simply destroy the todo.

That's all there is to it. Since we're binding to the change event, whenever the todo changes the view will automatically be re-rendered, checking or un-checking the checkbox as appropriate. Similarly, when the todo is destroyed, the model's `destroy()` function will be called, removing the todo from the view as we're binding to the destroy event too.

One more piece to mention is that we've also binded to a visible event to handle the visibility state of the todo item. This is used in conjunction with the filtering in our routes and collections so that we only display an item if its completed state falls in line with the current filter.

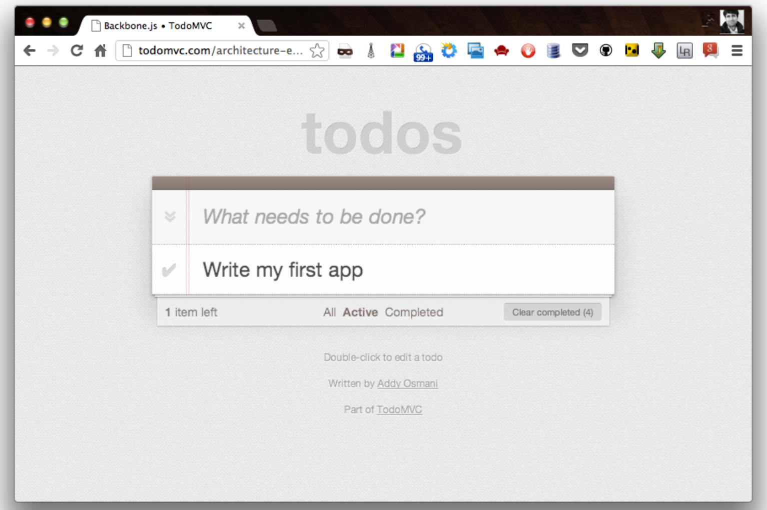
This tutorial is long enough as is, so we won't go into in-place editing or updating. If you want an example of that, see the complete source.

```
// js/view/todo.js

// Todo Item View
// --------------

// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName:  'li',

  // Cache the template function for a single item.
  template: _.template( $('#item-template').html() ),

  // The DOM events specific to an item.
  events: {
    'click .toggle':  'togglecompleted',
    'dblclick label': 'edit',
    'click .destroy': 'clear',
    'keypress .edit': 'updateOnEnter',
    'blur .edit':   'close'
  },

  // The TodoView listens for changes to its model, re-rendering. Since there's
  // a one-to-one correspondence between a **Todo** and a **TodoView** in this
  // app, we set a direct reference on the model for convenience.
  initialize: function() {
    this.model.on( 'change', this.render, this );
    this.model.on( 'destroy', this.remove, this );
    this.model.on( 'visible', this.toggleVisible, this );
  },

  // Re-render the titles of the todo item.
  render: function() {
    this.$el.html( this.template( this.model.toJSON() ) );
    this.$el.toggleClass( 'completed', this.model.get('completed') );

    this.toggleVisible();
    this.input = this.$('.edit');
    return this;
  },

  toggleVisible : function () {
    this.$el.toggleClass( 'hidden',  this.isHidden());
  },
  isHidden : function () {
    var isCompleted = this.model.get('completed');
    return ( // hidden cases only
      (!isCompleted && app.TodoFilter === 'completed')
      || (isCompleted && app.TodoFilter === 'active')
    );
  },

  // Toggle the `"completed"` state of the model.
  togglecompleted: function() {
    this.model.toggle();
  },

  // Switch this view into `"editing"` mode, displaying the input field.
  edit: function() {
    this.$el.addClass('editing');
    this.input.focus();
  },

  // Close the `"editing"` mode, saving changes to the todo.
  close: function() {
    var value = this.input.val().trim();

    if ( value ) {
      this.model.save({ title: value });
    } else {
      this.clear();
    }

    this.$el.removeClass('editing');
  },

  // If you hit `enter`, we're through editing the item.
  updateOnEnter: function( e ) {
    if ( e.which === ENTER_KEY ) {
      this.close();
    }
  },

  // Remove the item, destroy the model from *localStorage* and delete its view.
  clear: function() {
    this.model.destroy();
  }
});
```

Todo routing

Finally, we move on to routing, which will allow us to easily bookmark the list of items that are active as well as those which have been completed. We'll be supporting the following routes:

```
#/ (all - default)
#/active
#/completed
```



When the route changes the todo list will be filtered on a model level and the selected class on the filter links will be toggled. When an item is updated while in a filtered state, it will be updated accordingly. E.g. if the filter is active and the item is checked, it will be hidden. The active filter is persisted on reload.

```
// js/routers/router.js

// Todo Router
// ----------

var Workspace = Backbone.Router.extend({
  routes:{
    '*filter': 'setFilter'
  },

  setFilter: function( param ) {
    // Set the current filter to be used
    window.app.TodoFilter = param.trim() || '';

    // Trigger a collection filter event, causing hiding/unhiding
    // of Todo view items
    window.app.Todos.trigger('filter');
  }
});

app.TodoRouter = new Workspace();
Backbone.history.start();
```

As we can see in the line `window.app.Todos.trigger('filter')`, once a string filter has been set, we simply trigger our filter at a collection level to toggle which items are displayed and which of those are hidden.

Finally, we call `Backbone.history.start()` to route the initial URL during page load.

Conclusions

We've now learned how to build our first complete Backbone.js application. The full app can be viewed online at any time and the sources are readily available via TodoMVC.

Later on in the book, we'll learn how to further modularize this application using RequireJS, swap out our persistence layer to a database back-end and finally unit test the application with a few different testing frameworks.

Exercise 2: Book Library - Your first RESTful Backbone.js app

Credits: Björn Ekengren, Addy Osmani

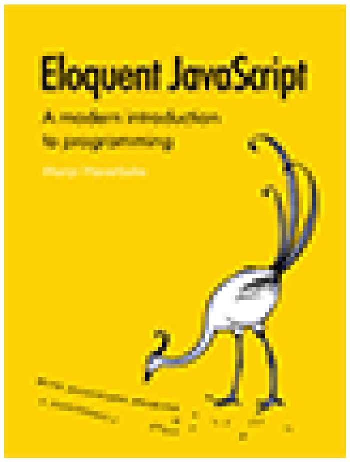
Part 1

In this exercise we will build a library application for managing digital books using Backbone.

Setting up

First we need a simple folder structure for our project. Create the directories css, img and js in your project root folder.

Download the Backbone, Underscore and jQuery libraries and copy them to your js folder. Save this image to your img folder:

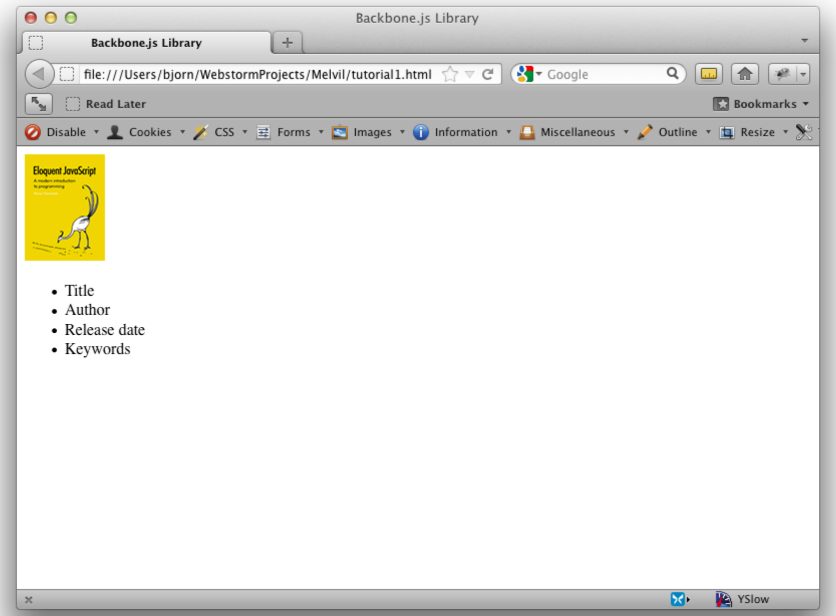


Create a new file index.html in the root of your project folder and enter this:

```html
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <title>Backbone.js Library</title>
    <link rel="stylesheet" href="css/screen.css">
</head>
<body>
<div id="books">
    <div class="bookContainer">
        <img src="img/placeholder.png"/>
        <ul>
            <li>Title</li>
            <li>Author</li>
            <li>Release date</li>
            <li>Keywords</li>
        </ul>
    </div>
</div>
<script src="js/jquery-1.7.1.js"></script>
<script src="js/underscore.js"></script>
```

```
<script src="js/backbone.js"></script>
<script src="js/app.js"></script>
</body>
</html>
```

open this file in a browser and it should look something like this:



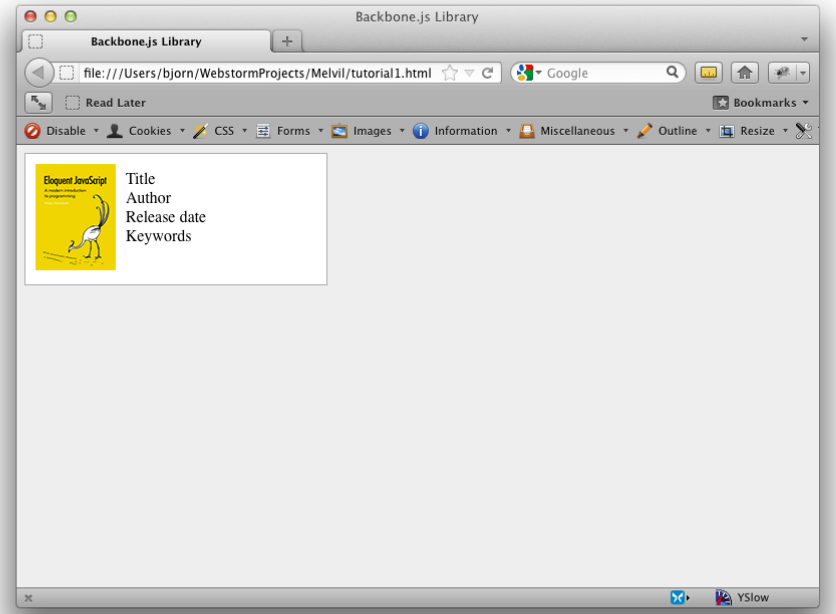
- Title
- Author
- Release date
- Keywords

Not so awesome. This is not a CSS tutorial, but we need to do some formatting. Create a file screen.css in your css folder and type this:

```
body {
    background-color: #eee;
}

.bookContainer {
    border: 1px solid #aaa;
    width: 300px;
    height: 130px;
    background-color: #fff;
    float: left;
    margin: 5px;
}
```

```
.bookContainer img {
    float: left;
    margin: 10px;
}
.bookContainer ul {
    list-style-type: none;
}
```

Now it looks a bit better:



Title
Author
Release date
Keywords

So this is what we want the final result to look like, but with more books. Go ahead and copy the bookContainer div a couple of times if you would like to see what it looks like. Now we are ready to start developing the actual application. Open up app.js and enter this:

```
(function ($) {

    var Book = Backbone.Model.extend({
        defaults:{
            coverImage:"img/placeholder.png",
            title:"Some title",
            author:"John Doe",
            releaseDate:"2012",
            keywords:"JavaScript Programming"
        }
    });

})(jQuery);
```

This is our model of a book. It contains a defaults property that gives us default values for all the values we want our model to contain. Note that I have wrapped everything in a self executing/invoking function with the argument "jQuery". This is standard procedure to prevent polluting the global scope and to avoid any collisions with the $ variable. The model itself isn't very useful so we need to pair it with a View. For the view we need a template so open up index.html and create a template from our bookContainer we created earlier:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8"/>
    <title>Backbone.js Library</title>
    <link rel="stylesheet" href="css/screen.css">
</head>
<body>
<div id="books">
    <div class="bookContainer">
        <img src="img/placeholder.png"/>
        <ul>
            <li>Title</li>
            <li>Author</li>
            <li>Release date</li>
            <li>Keywords</li>
        </ul>
    </div>
    <script id="bookTemplate" type="text/template">
        <img src="<%= coverImage %>"/>
        <ul>
            <li><%= title %></li>
            <li><%= author %></li>
            <li><%= releaseDate %></li>
            <li><%= keywords %></li>
        </ul>
    </script>
</div>
<script src="js/jquery-1.7.1.js"></script>
<script src="js/underscore.js"></script>
```

```
<script src="js/backbone.js"></script>
<script src="js/app.js"></script>
</body>
</html>
```

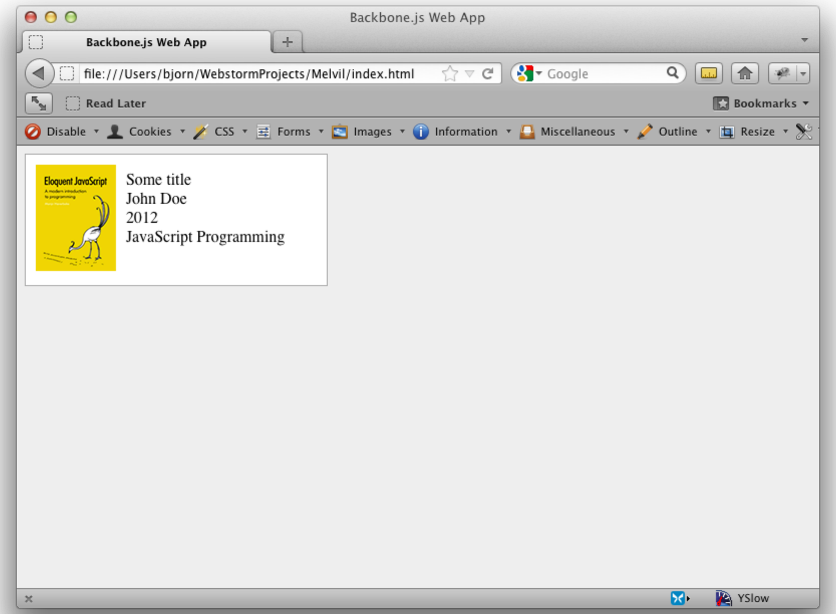
So what's going on here? Well, I have wrapped the template in a script tag with the type "text/template". By doing this the browser will not recognize the type and not render it, but we can still access all the elements in JavaScript. I have removed the surrounding div tag since the templating engine (Underscore) will create that for us. In the various fields I have inserted <%= field %> that Underscore will find and replace with real data. Now that we have our template in place we can go and create the view, so switch over to app.js:

```
(function ($) {

    var Book = Backbone.Model.extend({
        defaults:{
            coverImage:"img/placeholder.png",
            title:"Some title",
            author:"John Doe",
            releaseDate:"2012",
            keywords:"JavaScript Programming"
        }
    });

    var BookView = Backbone.View.extend({
        tagName:"div",
        className:"bookContainer",
        template:$("#bookTemplate").html(),

        render:function () {
            var tmpl = _.template(this.template); //tmpl is a function that takes a JSON ob

            this.$el.html(tmpl(this.model.toJSON())); //this.el is what we defined in tagNa
            return this;
        }
    });

})(jQuery);
```

So the view works like the model in that we use the extend function and pass it properties. The tagName property defines the container of the view, the className defines the class of the container and template is, well, the template for the view. The render function creates the tmpl function by calling underscores template function with one argument (the template from our index.html). It then calls the `tmpl` function with our model… wait a minute! What model? Well, this is something that we need to provide as an argument when we call our view constructor. As it looks now our view isn't connected to any element in our html page either, so let's take care of that now:

```
(function ($) {

    var Book = Backbone.Model.extend({
        defaults:{
            coverImage:"img/placeholder.png",
            title:"Some title",
            author:"John Doe",
            releaseDate:"2012",
            keywords:"JavaScript Programming"
        }
    });

    var BookView = Backbone.View.extend({
        tagName:"div",
        className:"bookContainer",
        template:$("#bookTemplate").html(),

        render:function () {
            var tmpl = _.template(this.template); //tmpl is a function that takes a JSON obj

            this.$el.html(tmpl(this.model.toJSON())); //this.el is what we defined in tagNa
            return this;
        }
    });

    var book = new Book({
        title:"Some title",
        author:"John Doe",
        releaseDate:"2012",
        keywords:"JavaScript Programming"
    });

    bookView = new BookView({
        model:book
    });

    $("#books").html(bookView.render().el);

})(jQuery);
```

Here we have created a new Book model by calling the Book constructor and passing an object with our desired properties. The model is then used when creating a BookView. Finally the bookView is rendered and inserted into our page. It should look something like this:



Some title
John Doe
2012
JavaScript Programming

Looking good, we now have a working Backbone application. Since a library cannot contain just one book, we need to push further. Lets have a look at Collections. Collections contain collections of models. The models must be of the same kind, i.e. you cannot mix apples and oranges in the same collection. Other than that, collections are quite simple, you just tell them what kind of models they contain like this:

```
var Library = Backbone.Collection.extend({
    model:Book
});
```

Go ahead and insert the code in app.js. Next stop is a new view for our Library collection. This view is a bit more complicated than the earlier BookView:

```
var LibraryView = Backbone.View.extend({
    el:$("#books"),

    initialize:function(){
        this.collection = new Library(books);
        this.render();
    },

    render:function(){
        var that = this;
        _.each(this.collection.models, function(item){
            that.renderBook(item);
        }, this);
    },

    renderBook:function(item){
        var bookView = new BookView({
            model:item
        });
        this.$el.append(bookView.render().el);
    }
});
```

I added some line numbers to help me explain what is going on. On line 02 I specified the property "el". A view can take either a tagName, as we saw in our `BookView`, or an el. When we use tagName, the view will create the element for us, but we are responsible for inserting it into the page. When we use el we specify an existing element in the page and the view will write directly into the page, into the specified element. In this case we select the div with id="books".

Next on line 04 is the initialize function. This function will be called by Backbone when the view constructor is called. On line 05 in we create a new Library (collection of Book models) and store it in a local property called "collection". On line 06 it calls its own render function, which means that as soon as we call the `LibraryView` constructor it will get rendered, so this is a self rendering view. We don't have to make it self rendered but it is common practice.

On line 10 in the render function we store a reference to the current object in a variable "that" and then (on line 11) use the each function of underscore to iterate over all the models (Books) in our collection. The first argument to "each" is the array that will be iterated over. The second argument is the function that will be applied to each member of the array. The function in our case calls the renderBook function with the current model as argument. We need to use "that" to get this right since if we would have used "this" it would have referenced the function itself. The third argument (line 13) is the context that each is executing in.

On line 16 we define a function `renderBook` that takes a model (a Book) as argument and uses it to create a `BookView`. The `bookView` is then rendered and appended to the view container as specified in our el property (on line 02).

You may have noticed that on line 05 we called the Library constructor with the argument “books”. Library is a Backbone collection that expects an array of objects that it can use to create Book models. We haven’t defined the books variable yet so lets go ahead and do that:

```
var books = [{title:"JS the good parts", author:"John Doe", releaseDate:"2012", keywords:"Ja
    {title:"CS the better parts", author:"John Doe", releaseDate:"2012", keywords:"CoffeeScr
    {title:"Scala for the impatient", author:"John Doe", releaseDate:"2012", keywords:"Scala
    {title:"American Psyco", author:"Bret Easton Ellis", releaseDate:"2012", keywords:"Novel
    {title:"Eloquent JavaScript", author:"John Doe", releaseDate:"2012", keywords:"JavaScrip
```

Now we are almost ready to try out our first version of the Backbone library. Replace this code:

```
var book = new Book({
    title:"Some title",
    author:"John Doe",
    releaseDate:"2012",
    keywords:"JavaScript Programming"
});

bookView = new BookView({
    model:book
});

$("#books").html(bookView.render().el);
```

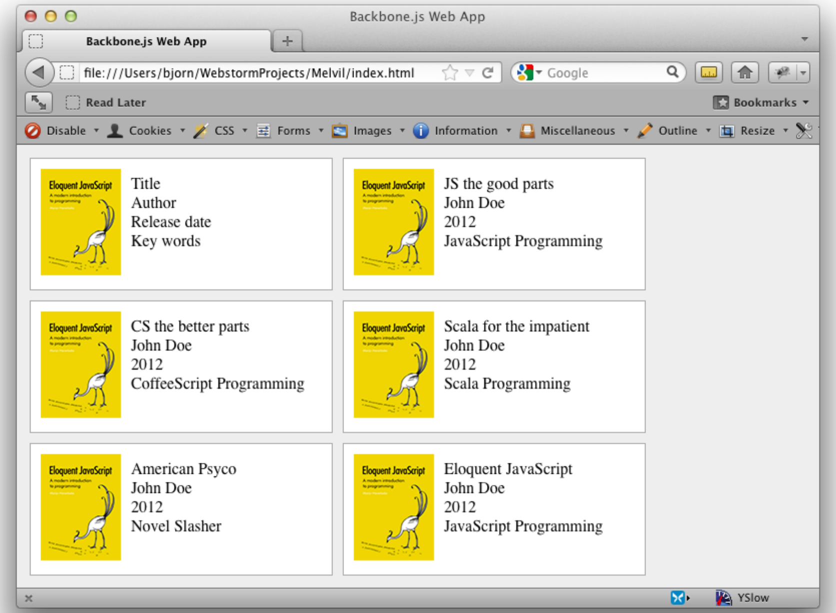
with this:

```
var libraryView = new LibraryView();
```

If you view index.html in a browser you should see something like this:

Here is the final app.js:

```
(function ($) {
    var books = [{title:"JS the good parts", author:"John Doe", releaseDate:"2012", keywords
        {title:"CS the better parts", author:"John Doe", releaseDate:"2012", keywords:"Coffe
        {title:"Scala for the impatient", author:"John Doe", releaseDate:"2012", keywords:"S
        {title:"American Psyco", author:"Bret Easton Ellis", releaseDate:"2012", keywords:"N
        {title:"Eloquent JavaScript", author:"John Doe", releaseDate:"2012", keywords:"JavaS

    var Book = Backbone.Model.extend({
        defaults:{
            coverImage:"img/placeholder.png",
```



Title
Author
Release date
Key words

JS the good parts
John Doe
2012
JavaScript Programming

CS the better parts
John Doe
2012
CoffeeScript Programming

Scala for the impatient
John Doe
2012
Scala Programming

American Psyco
John Doe
2012
Novel Slasher

Eloquent JavaScript
John Doe
2012
JavaScript Programming

```
            title:"Some title",
            author:"John Doe",
            releaseDate:"2012",
            keywords:"JavaScript Programming"
        }
});

var Library = Backbone.Collection.extend({
   model:Book
});

var BookView = Backbone.View.extend({
    tagName:"div",
    className:"bookContainer",
    template:$("#bookTemplate").html(),

    render:function () {
        var tmpl = _.template(this.template); //tmpl is a function that takes a JSON ob

        this.$el.html(tmpl(this.model.toJSON())); //this.el is what we defined in tagNa
        return this;
        }
    });

 var LibraryView = Backbone.View.extend({
        el:$("#books"),

        initialize:function(){
            this.collection = new Library(books);
            this.render();
        },

        render:function(){
            var that = this;
            _.each(this.collection.models, function(item){
                that.renderBook(item);
            }, this);
        },

        renderBook:function(item){
            var bookView = new BookView({
                model:item
            });
            this.$el.append(bookView.render().el);
        }
    });

    var libraryView = new LibraryView();

})(jQuery);
```

So lets recap what happens here:

- Line 58: the LibraryView constructor is called
- Line 36: the LibraryView is tied to the books div in our html
- Line 39: the Library constructor is called with the array of book properties from line 02, creating a collection of Book models
- Line 43: The render function iterates over the Book collection and calls renderBook (line 46) for each Book model
- Line 51: renderBook function creates a new BookView from the given Book model
- Line 54: The render function of BookView is called and the result is appended to the books div in our html

You have now been introduced to the basic components of Backbone Model, View and Collection. In the next part I will continue to develop our Library into something more interactive.

The code for this tutorial is available here.

Part 2

In the first part, I covered some Backbone basics of models, views and collections. I recommend reading that before continuing since this tutorial builds on the first part. In this tutorial we are going to look at how we can add and remove models from a collection.

Adding models

To add a new book we need to have some sort of input form. Go ahead and add one in index.html

```
<div id="books">
    <form id="addBook" action="#">
        <div>
            <label for="coverImage">CoverImage: </label><input id="coverImage" type="file" /
            <label for="title">Title: </label><input id="title" type="text" />
            <label for="author">Author: </label><input id="author" type="text" />
            <label for="releaseDate">Release date: </label><input id="releaseDate" type="tex
            <label for="keywords">Keywords: </label><input id="keywords" type="text" />
            <button id="add">Add</button>
        </div>
    </form>
</div>
```

I made the ids of the inputs match the attributes in our Book model so that we don't have to remap those later. Add this to screen.css for looks

```
#addBook label {
    width:100px;
    margin-right:10px;
    text-align:right;
    line-height:25px;
}

#addBook label, #addBook input {
    display:block;
    margin-bottom:10px;
    float:left;
}
```

```
#addBook label[for="title"], #addBook label[for="releaseDate"] {
    clear:both;
}

#addBook button {
    display:block;
    margin:5px 20px 10px 10px;
    float:right;
    clear:both;
}

#addBook div {
    width:550px;
}

#addBook div:after {
    content:"";
    display:block;
    height:0;
    visibility:hidden;
    clear:both;
    font-size:0;
    line-height:0;
}
```

Now lets create a function that lets us add a book. We put it in our master view (LibraryView) after the render function

```
addBook:function(){
    var formData = {};

    $("#addBook div").children("input").each(function(i, el){
        formData[el.id] = $(el).val();
    });

    books.push(formData);

    this.collection.add(new Book(formData));
},
```

Here I select all the input elements of the form and iterate over them using jQuery's each. Since we used the same names for ids in our form as the keys on our Book model we can simply store them directly in the formData object and add it to the books array. We then create a new Book model and add it to our collection. Now lets wire this function to the Add button in our form. We do this by adding an event listener in the LibraryView

```
events:{
    "click #add":"addBook"
},
```

By default, Backbone will send an event object as parameter to the function. This is useful for us in this case since we want to prevent the form from actually submit and reloading the page. Add a preventDefault to the addBook function

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

    var formData = {};

    $("#addBook div").children("input").each(function(i, el){
        formData[el.id] = $(el).val();
    });

    books.push(formData);

    this.collection.add(new Book(formData));
},
```

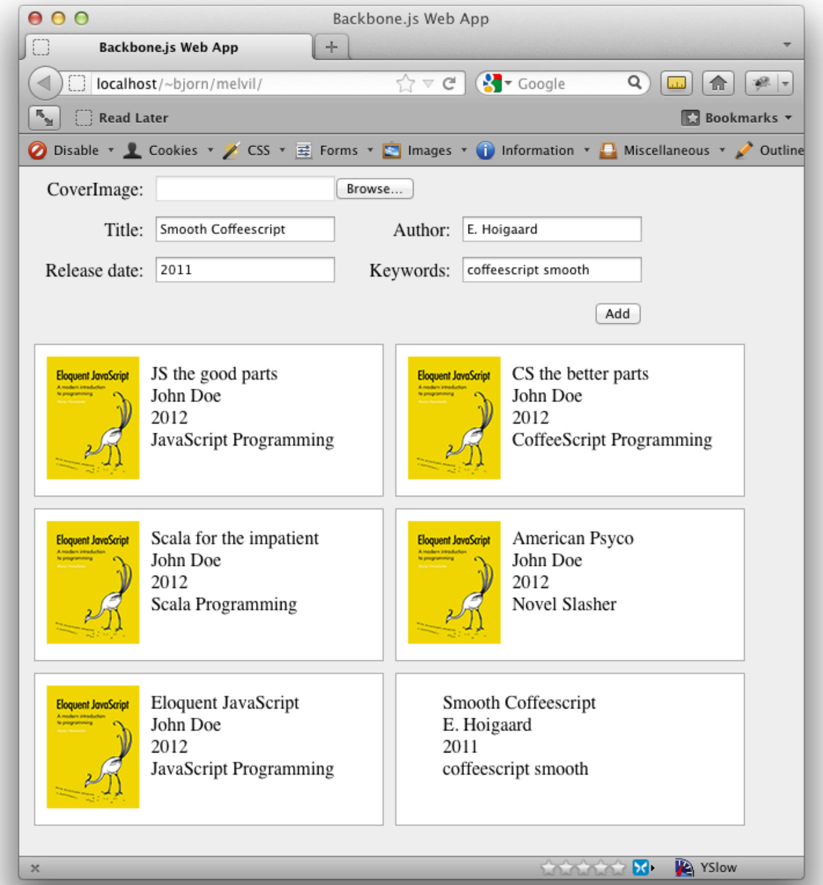
Ok, so this will add a book to our books array and our collection. It will not however be visible, because we have not called the render function of our BookView. When a model is added to a collection, the collection will fire an add event. If we listen to this event we will be able to call our renderBook function. This binding is done in initialize.

```
initialize:function () {
    this.collection = new Library(books);
    this.render();

    this.collection.on("add", this.renderBook, this);
},
```

Now you should be ready to take the application for a spin.

As you may notice, if you leave a field blank, it will be blank in the created view as well. This is not what we want, we would like the default values to kick in. To do that we need to add a bit of logic. Also note that the file input for the cover image isn't working, but that is left as an exercise to the reader.

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



CoverImage:			
Title:	Smooth Coffeescript	Author:	E. Hoigaard
Release date:	2011	Keywords:	coffeescript smooth

JS the good parts
John Doe
2012
JavaScript Programming

CS the better parts
John Doe
2012
CoffeeScript Programming

Scala for the impatient
John Doe
2012
Scala Programming

American Psyco
John Doe
2012
Novel Slasher

Eloquent JavaScript
John Doe
2012
JavaScript Programming

Smooth Coffeescript
E. Hoigaard
2011
coffeescript smooth

```
    var formData = {};

    $("#addBook div").children("input").each(function (i, el) {
        if ($(el).val() !== "") {
            formData[el.id] = $(el).val();
        }
    });

    books.push(formData);

    this.collection.add(new Book(formData));
},
```

Here I added (line 7) a check to see if the field value is empty, in which case we do not add it to the model data. While we are at it lets add default values for the other properties of Book

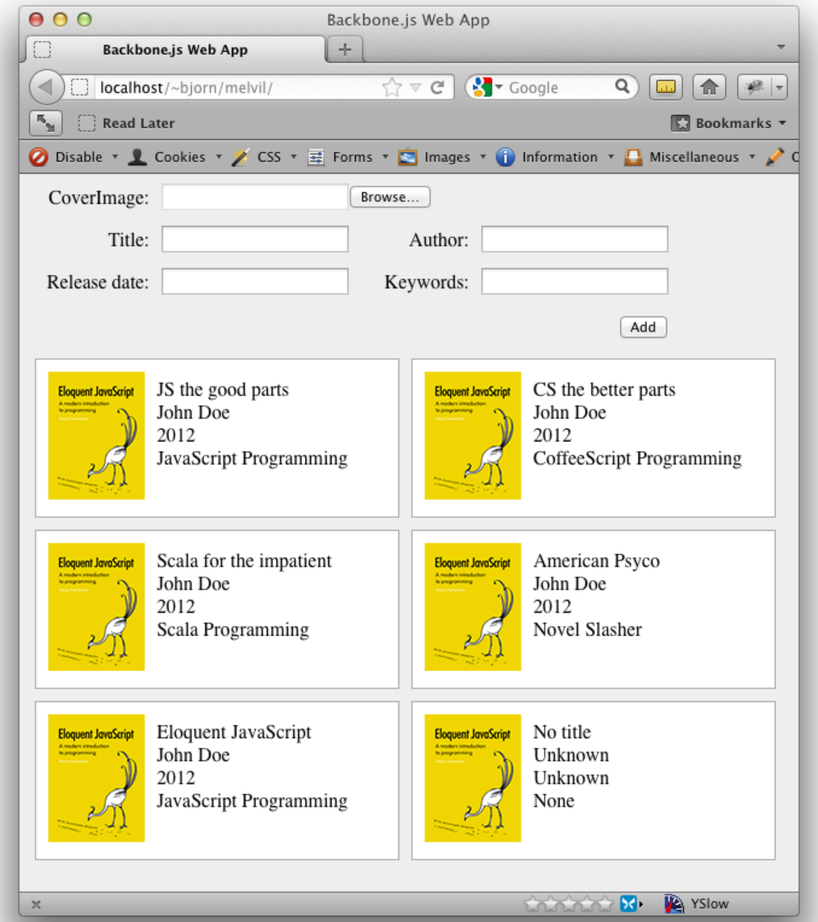
```
var Book = Backbone.Model.extend({
    defaults:{
        coverImage:"img/placeholder.png",
        title:"No title",
        author:"Unknown",
        releaseDate:"Unknown",
        keywords:"None"
    }
});
```

Now it has better default behaviour

Removing models

Now lets see how to remove Books. We start by adding a delete button to the template in index.html

```
<script id="bookTemplate" type="text/template">
    <img src="<%= coverImage %>" />
    <ul>
        <li><%= title %></li>
        <li><%= author %></li>
        <li><%= releaseDate %></li>
        <li><%= keywords %></li>
    </ul>

    <button class="delete">Delete</button>
</script>
```



CoverImage:

Title: Author:

Release date: Keywords:

JS the good parts
John Doe
2012
JavaScript Programming

CS the better parts
John Doe
2012
CoffeeScript Programming

Scala for the impatient
John Doe
2012
Scala Programming

American Psyco
John Doe
2012
Novel Slasher

Eloquent JavaScript
John Doe
2012
JavaScript Programming

No title
Unknown
Unknown
None

We add some css to it for good looks. Note that I removed the margin of the existing ul rule above to tighten things up a bit.

```
.bookContainer ul {
    list-style-type:none;
    margin-bottom:0;
}

.bookContainer button {
    float:right;
    margin:10px;
}
```

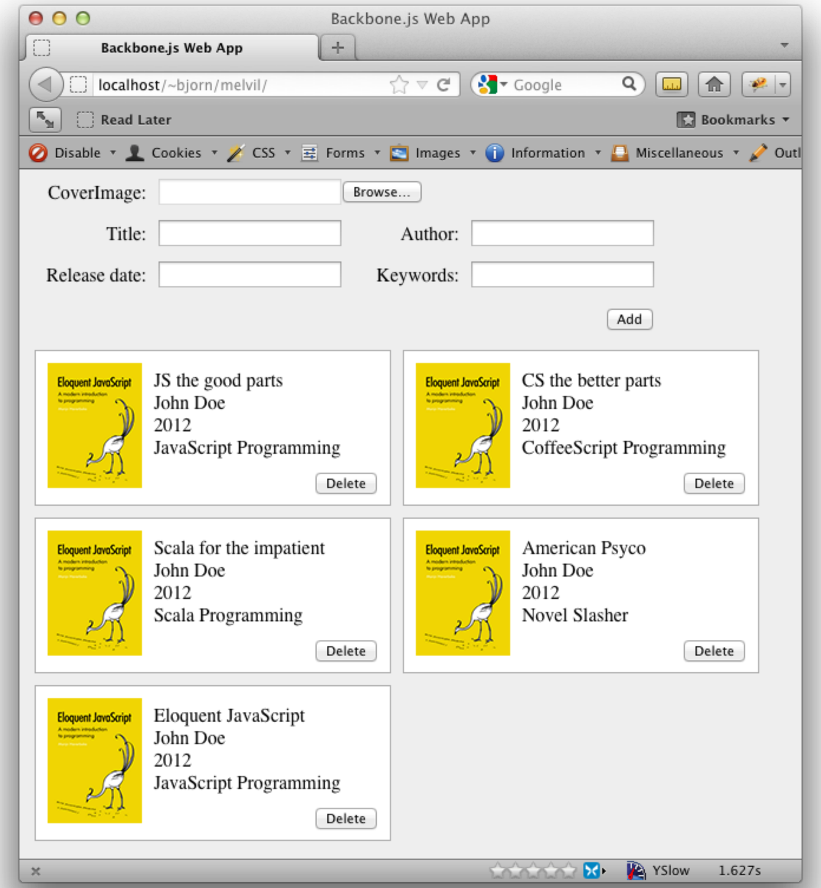
Looks okay!

Now we need to wire up the button to the logic. This works in the same way as with add. We start by creating a deleteBook function in the BookView

```
var BookView = Backbone.View.extend({
    tagName:"div",
    className:"bookContainer",
    template:$("#bookTemplate").html(),

    render:function () {
        var tmpl = _.template(this.template); //tmpl is a function that takes a JSON and re

        this.$el.html(tmpl(this.model.toJSON())); //this.el is what we defined in tagName.
        return this;
    },

    deleteBook:function () {
        //Delete model
        this.model.destroy();

        //Delete view
        this.remove();
    }
});
```

Then we add an event listener to the delete button

```
var BookView = Backbone.View.extend({
    tagName:"div",
    className:"bookContainer",
    template:$("#bookTemplate").html(),
```



CoverImage:			
Title:		Author:	
Release date:		Keywords:	

JS the good parts
John Doe
2012
JavaScript Programming

CS the better parts
John Doe
2012
CoffeeScript Programming

Scala for the impatient
John Doe
2012
Scala Programming

American Psyco
John Doe
2012
Novel Slasher

Eloquent JavaScript
John Doe
2012
JavaScript Programming

```
    render:function () {
        var tmpl = _.template(this.template); //tmpl is a function that takes a JSON and re

        this.$el.html(tmpl(this.model.toJSON())); //this.el is what we defined in tagName.
        return this;
    },

    events: {
        "click .delete":"deleteBook"
    },

    deleteBook:function () {
        //Delete model
        this.model.destroy();

        //Delete view
        this.remove();
    }
});
```

If you try it out now you will see that it seems to work. However we are not yet finished. When we click the delete button the model and view will be deleted, but not the data in our books array. This means that if we were to re-render the LibraryView the deleted books would reappear. The Backbone collection is smart enough to notice when one of its models is deleted and will fire a "remove" event. This is something we can listen to in our LibraryView and take action. Add a listener in initialize of LibraryView

```
initialize:function () {
    this.collection = new Library(books);
    this.render();

    this.collection.on("add", this.renderBook, this);
    this.collection.on("remove", this.removeBook, this);
},
```

Here I specified that the removeBook function should be called when the remove event from our collection fires, so lets create this function. Note that the collection provides the removed model as a parameter to the event.

```
removeBook:function(removedBook){
    var removedBookData = removedBook.attributes;

    _.each(removedBookData, function(val, key){
        if(removedBookData[key] === removedBook.defaults[key]){
            delete removedBookData[key];
        }
    });

    _.each(books, function(book){
        if(_.isEqual(book, removedBookData)){
            books.splice(_.indexOf(books, book), 1);
        }
    });
},
```

Since the default values were not saved in the books array we need to remove them in order to find a match. We use underscores each function to iterate over the properties of the removed Book model and delete any property that is equal to the default value. Since the underscore each function is also capable of iterating over objects in an array we use it again to iterate over the objects in our books array to find the data of the removed Book. To get a match we use the isEqual function of underscore that performs a deep comparison of objects. Similarly the indexOf can find complex objects, which we use when we remove the book data using splice.

Summary

In this tutorial we have looked at how to add and remove models from a collection. We also looked at how to wire view, model and collection together using event handlers. We also saw the useful underscore functions each, indexOf and isEqual.

The code for this part is available here.

Part 2.5

In the first two parts of this tutorial series we looked at the basic structure of a Backbone application and how to add and remove models. In the third part we will look at how to synchronize the models with the back end, but in order to do that we need to make a small detour and set up a server with a REST api. That is what we are going to do in this part. Since this is a JavaScript tutorial I will use JavaScript to create the server using node.js. If you are more comfortable in setting up a REST server in another language, this is the API you need to conform to:

```
url HTTP Method Operation
/api/books  GET Get an array of all books
/api/books/:id  GET Get the book with id of :id
```

```
/api/books  POST  Add a new book and return the book with an id attribute added
/api/books/:id  PUT Update the book with id of :id
/api/books/:id  DELETE  Delete the book with id of :id
```

The outline for this tutorial looks like this:

- Install node.js, npm and MongoDB
- Install node modules
- Create directory structure
- Create a simple web server
- Connect to the database
- Create the REST API

Download and install node.js from nodejs.org. The node package manager (npm) will be installed as well.

Download and install MongoDB from mongodb.org. When installing MongoDB you get instructions on copying a config file in order for MongoDB to run. On OSX it can look something like this:

```
sudo cp /usr/local/Cellar/mongodb/2.0.4-x86_64/mongod.conf /usr/local/etc/mongod.conf
```

Once it is installed we can go ahead and start it. On my machine I did this:

```
mongod run --config /usr/local/Cellar/mongodb/2.0.4-x86_64/mongod.conf
```

Installing node modules

Start by creating a new folder for this project. Using a terminal, go to your project folder and run the following commands:

```
npm install express@2.5.9
npm install path
npm install mongoose
```

Note the @2.5.9 on express. The current stable is 3.0.3 and has a different API, but we will be using 2.5.9 for the time being.

Create directory structure

In your project folder root, create a file server.js – this is where our server code will go. Then create a folder public. Anything within the public folder will be served by the express web server as static content to the client. Now go ahead and copy everything from Part 2 into the public folder. When you are done your folder structure should look something like this:

```
node_modules/
  .bin/
  express/
  mongoose/
  path/
public/
  css/
  img/
  js/
  index.html
server.js
package.json
```

Create a simple web server

Open server.js and enter the following:

```
// Module dependencies.
var application_root = __dirname,
    express = require("express"), //Web framework
    path = require("path"), //Utilities for dealing with file paths
    mongoose = require('mongoose'); //MongoDB integration

//Create server
var app = express.createServer();

// Configure server
app.configure(function () {
    app.use(express.bodyParser()); //parses request body and populates req.body
    app.use(express.methodOverride()); //checks req.body for HTTP method overrides
    app.use(app.router); //perform route lookup based on url and HTTP method
    app.use(express.static(path.join(application_root, "public"))); //Where to serve static
    app.use(express.errorHandler({ dumpExceptions:true, showStack:true })); //Show all erro
});

//Start server
app.listen(4711, function () {
    console.log("Express server listening on port %d in %s mode", app.address().port, app.se
});
```

I start off by loading the modules required for this project: Express for creating the HTTP server, Path for dealing with file paths and mongoose for connecting with the database. We then create an express server and configure it using an anonymous function. This is a pretty standard configuration and for our application we don't actually need the methodOverride part. It is used for issuing PUT and DELETE HTTP requests directly from a form, since forms normally only support GET and POST. Finally I start the server by running the listen function. The port number used, in this case 4711, could be any free port on your system. I simply used 4711 since it is the most random number. We are now ready to run our first server:

```
node server.js
```

If you open a browser on localhost:4711 you should see something like this:

This is where we left off in Part 2, but we are now running on a server instead of directly from the files. Great job! We can now start defining routes (URLs) that the server should react to. This will be our REST API. Routes are defined by using app followed by one of the HTTP verbs get, put, post and delete, which corresponds to Create, Read, Update and Delete. Let us go back to server.js and define a simple route:

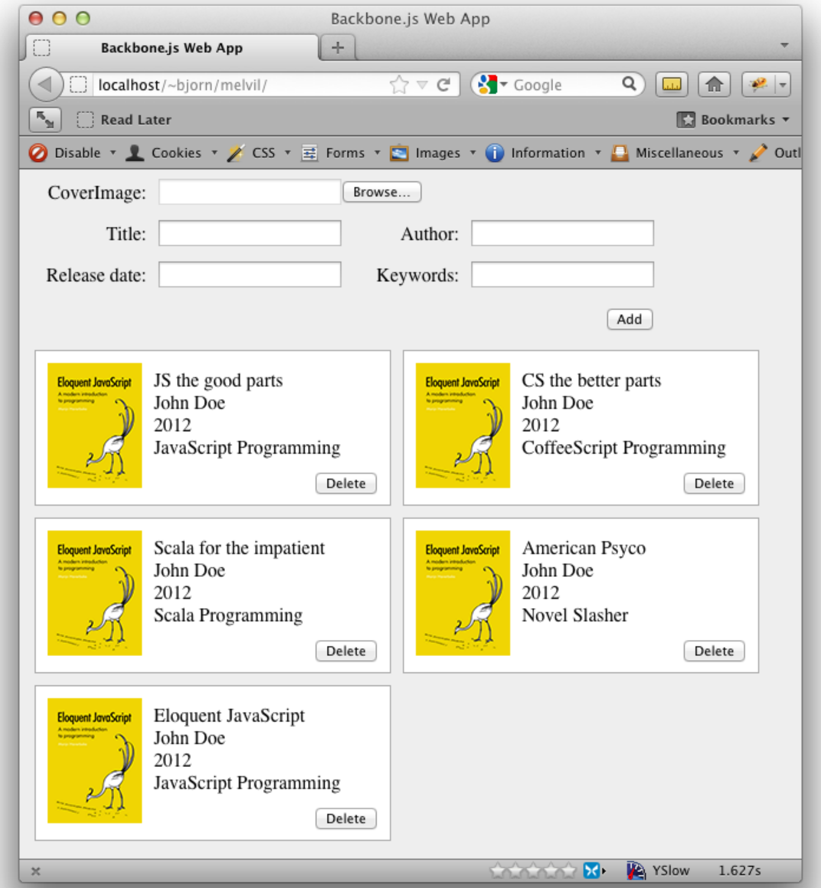
```
// Routes
app.get('/api', function(req, res){
    res.send('Library API is running');
});
```

The get function will take the URL as first parameter and a function as second. The function will be called with request and response objects. Now you can restart node and go to our specified URL:

Connect to database

Fantastic. Now since we want to store our data in MongoDB we need to define a schema. Add this to server.js:

```
//Connect to database
mongoose.connect('mongodb://localhost/library_database');

//Schemas
var Book = new mongoose.Schema({
```



CoverImage:

Title:		Author:	
Release date:		Keywords:	

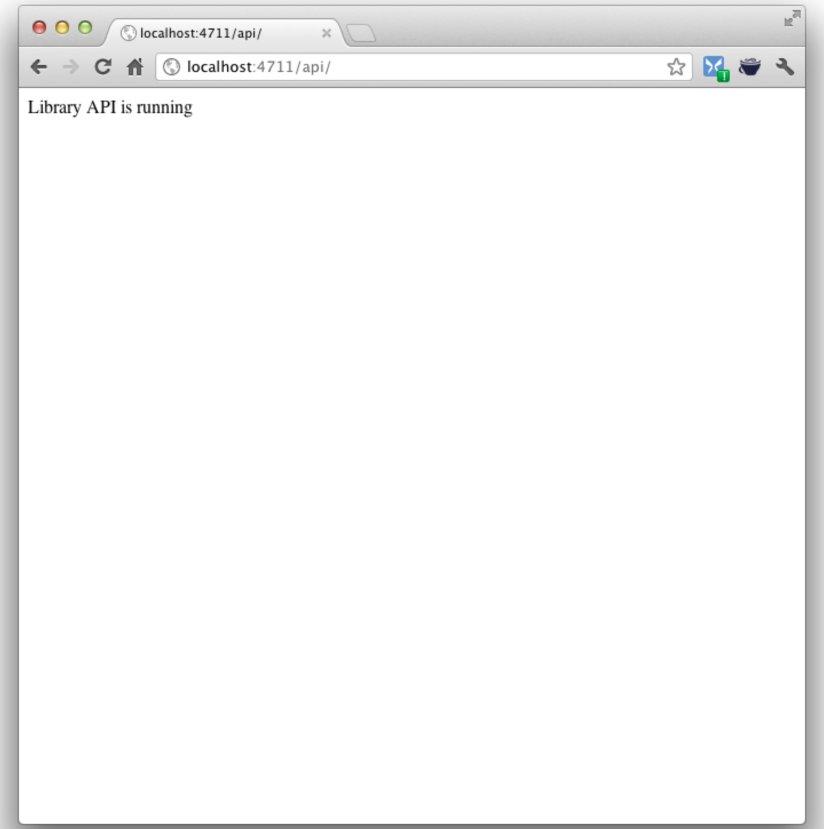
JS the good parts
John Doe
2012
JavaScript Programming

CS the better parts
John Doe
2012
CoffeeScript Programming

Scala for the impatient
John Doe
2012
Scala Programming

American Psyco
John Doe
2012
Novel Slasher

Eloquent JavaScript
John Doe
2012
JavaScript Programming



```
Library API is running
```

```
    title:String,
    author:String,
    releaseDate:Date
});

//Models
var BookModel = mongoose.model('Book', Book);
```

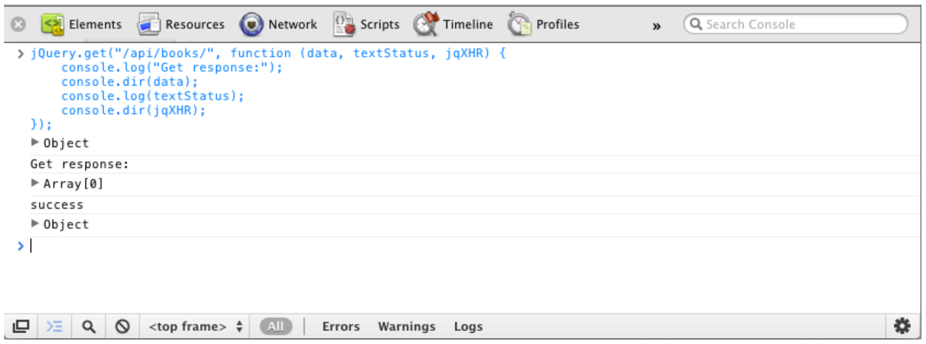
As you can see, schema definitions are quite straight forward. They can be more advanced, but this will do for us. I also extracted a model (BookModel) from Mongo. This is what we will be working with. Next up we define a get operation for the REST API that will return all books:

```
//Get a list of all books
app.get('/api/books', function (req, res) {
    return BookModel.find(function (err, books) {
        if (!err) {
            return res.send(books);
        } else {
            return console.log(err);
        }
    });
});
```

The find function of Model is defined like this: function find (conditions, fields, options, callback) – but since we want a function that return all books we only need the callback parameter. The callback will be called with an error object and an array of found objects. If there was no error we return the array of objects to the client using the send function of the result object, otherwise we log the error to the console.

To test our API we need to do a little typing in a JavaScript console. Restart node and go to localhost:4711 in your browser. Open up the JavaScript console. If you are using Google Chrome, go to View->Developer->JavaScript Console. If you are using Firefox, install Firebug and go to View->Firebug. If you are on any other browser I'm sure you will find a console somewhere. In the console type the following:

```
jQuery.get("/api/books/", function (data, textStatus, jqXHR) {
    console.log("Get response:");
    console.dir(data);
    console.log(textStatus);
    console.dir(jqXHR);
});
```



```
> jQuery.get("/api/books/", function (data, textStatus, jqXHR) {
      console.log("Get response:");
      console.dir(data);
      console.log(textStatus);
      console.dir(jqXHR);
  });
  ▶ Object
  Get response:
  ▶ Array[0]
  success
  ▶ Object
> |
```

. . . and press enter and you should get something like this:

Here I used jQuery to make the call to our REST API, since it was already loaded on the page. The returned array is obviously empty, since we have not put anything into the database yet. Lets go and create a POST route that enables this in server.js:

```
//Insert a new book
app.post('/api/books', function (req, res) {
    var book = new BookModel({
        title:req.body.title,
        author:req.body.author,
        releaseDate:req.body.releaseDate
    });
    book.save(function (err) {
        if (!err) {
            return console.log('created');
        } else {
            return console.log(err);
        }
    });
    return res.send(book);
});
```

We start by creating a new BookModel passing an object with title, author and releaseDate attributes. The data are collected from req.body. This means that anyone calling this operation in the API needs to supply a JSON object containing the title, author and releaseDate attributes. Actually, the caller can omit any or all attributes since we have not made any one mandatory.

We then call the save function on the BookModel passing in an anonymous function for handling errors in the same way as with the previous get route. Finally we return the saved BookModel. The reason we return the BookModel and not just "success" or similar string is that when the BookModel is saved it will get an _id attribute from MongoDB, which the client needs when updating or deleting a specific book. Lets try it out again, restart node and go back to the console and type:

```
jQuery.post("/api/books", {
  "title":"JavaScript the good parts",
  "author":"Douglas Crockford",
  "releaseDate":new Date(2008, 4, 1).getTime()
}, function(data, textStatus, jqXHR) {
    console.log("Post response:");
    console.dir(data);
    console.log(textStatus);
    console.dir(jqXHR);
});
```

..and then

```
jQuery.get("/api/books/", function (data, textStatus, jqXHR) {
    console.log("Get response:");
    console.dir(data);
    console.log(textStatus);
    console.dir(jqXHR);
});
```

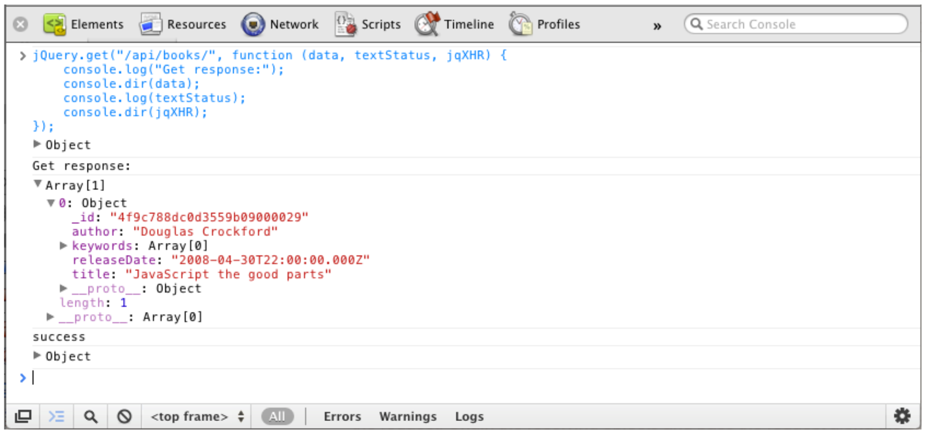
You should now get an array of size 1 back from our server. You may wonder about this line:

```
"releaseDate":new Date(2008, 4, 1).getTime()
```

MongoDB expects dates in UNIX time format (milliseconds from Jan 1st 1970), so we have to convert dates before posting. The object we get back however, contains a JavaScript Date object. Also note the _id attribute of the returned object.

Lets move on to creating a get that retrieves a single book in server.js:

```
app.get('/api/books/:id', function(req, res){
    return BookModel.findById(req.params.id, function(err, book){
        if(!err){
            return res.send(book);
        } else {
            return console.log(err);
        }
    });
});
```



```
> jQuery.get("/api/books/", function (data, textStatus, jqXHR) {
      console.log("Get response:");
      console.dir(data);
      console.log(textStatus);
      console.dir(jqXHR);
  });
  ▶ Object
  Get response:
  ▼ Array[1]
    ▼ 0: Object
        _id: "4f9c788dc0d3559b09000029"
        author: "Douglas Crockford"
      ▶ keywords: Array[0]
        releaseDate: "2008-04-30T22:00:00.000Z"
        title: "JavaScript the good parts"
      ▶ __proto__: Object
      length: 1
    ▶ __proto__: Array[0]
  success
  ▶ Object
> |
```

Here we use colon notation (:id) to tell express that this part of the route is dynamic. We also use the findById function on BookModel to get a single result. Now you can get a single book by adding its id to the URL like this:

```
jQuery.get("/api/books/4f95a8cb1baa9b8a1b000006", function (data, textStatus, jqXHR){
    console.log("Get response:");
    console.dir(data);
    console.log(textStatus);
    console.dir(jqXHR);
});
```

Lets create the PUT (update) function next:

```
app.put('/api/books/:id', function(req, res){
    console.log('Updating book ' + req.body.title);
    return BookModel.findById(req.params.id, function(err, book){
        book.title = req.body.title;
        book.author = req.body.author;
        book.releaseDate = req.body.releaseDate;
        return book.save(function(err){
            if(!err){
                console.log('book updated');
            } else {
                console.log(err);
            }
            return res.send(book);
        });
    });
});
```

This is a little larger than previous ones, but should be pretty straight forward – we find a book by id, update its properties, save it and send it back to the client.

To test this we need to use the more general jQuery ajax function (replace the id with what you got from a GET request):

```
jQuery.ajax({
  url:"/api/books/4f95a8cb1baa9b8a1b000006",
  type:"PUT",
  data:{
    "title":"JavaScript The good parts",
    "author":"The Legendary Douglas Crockford",
    "releaseDate":new Date(2008, 4, 1).getTime()
  },
  success: function(data, textStatus, jqXHR) {
    console.log("Post response:");
    console.dir(data);
    console.log(textStatus);
    console.dir(jqXHR);
  }
});
```

Finally we create the delete route:

```
app.delete('/api/books/:id', function(req, res){
    console.log('Deleting book with id: ' + req.params.id);
    return BookModel.findById(req.params.id, function(err, book){
        return book.remove(function(err){
            if(!err){
                console.log('Book removed');
                return res.send('');
            } else {
                console.log(err);
            }
        });
    });
});
```

. . . and try it out:

```
jQuery.ajax({
  url:'/api/books/4f95a5251baa9b8a1b000001',
  type:'DELETE',
  success:function(data, textStatus, jqXHR){
    console.log("Post response:");
    console.dir(data);
    console.log(textStatus);
    console.dir(jqXHR);
  }
});
```

So now our REST API is complete – we have support for all HTTP verbs. What next? Well, until now I have left out the keywords part of our books. This is a bit more complicated since a book could have several keywords and we don't want to represent them as a string, but rather an array of strings. To do that we need another schema. Add a Keywords schema right above our Book schema:

```
//Schemas
var Keywords = new mongoose.Schema({
    keyword:String
});
```

To add a sub schema to an existing schema we use brackets notation like so:

```
var Book = new mongoose.Schema({
    title:String,
    author:String,
    releaseDate:Date,
    keywords:[Keywords]
});
```

Also update POST and PUT

```
app.post('/api/books', function (req, res) {
    var book = new BookModel({
        title:req.body.title,
        author:req.body.author,
        releaseDate:req.body.releaseDate,
        keywords:req.body.keywords
    });
    book.save(function (err) {
        if (!err) {
            return console.log('created');
        } else {
            return console.log(err);
        }
    });
    return res.send(book);
});
```

```
app.put('/api/books/:id', function(req, res){
    return BookModel.findById(req.params.id, function(err, book){
        book.title = req.body.title;
        book.author = req.body.author;
        book.releaseDate = req.body.releaseDate;
        book.keywords = req.body.keywords;
        return book.save(function(err){
            if(!err){
                console.log('book updated');
            } else {
                console.log(err);
            }
            return res.send(book);
        });
    });
});
```

There we are, that should be all we need, now we can try it out in the console:

```
jQuery.post("/api/books", {
  "title":"Secrets of the JavaScript Ninja",
  "author":"John Resig",
  "releaseDate":new Date(2008, 3, 12).getTime(),
  "keywords":[{
     "keyword":"JavaScript"
   },{
     "keyword":"Reference"
   }]
}, function(data, textStatus, jqXHR) {
    console.log("Post response:");
    console.dir(data);
    console.log(textStatus);
    console.dir(jqXHR);
});
```

You should now have a fully functional REST server. We will use this in the next part of the tutorial to make our library application persistent.

Part 3

In this part we will cover connecting our Backbone application to a server through a REST API. This tutorial builds on the first two parts, so I strongly recommend reading those to fully benefit from this part.

To synchronize a Backbone application to a server we need a server with a REST API that communicates in JSON format. The server I created in part 2.5 will fit our needs, but if you are creating your own server in another language you might want to read through that tutorial to get a grip of the API. Backbone makes use of the sync function in order to persist models with the server. You usually do not need to use this function directly, but instead set a url attribute to a collection (or a Model if the Models are not in a collection) informing Backbone where to sync. So lets go ahead and add a url attribute to our Library collection:

```
var Library = Backbone.Collection.extend({
    model:Book,
    url:'/api/books'
});
```

Hint: If you, for some reason, don't have access to a server you could use the Backbone.Localstorage plugin (you'll need a browser that supports localstorage though). To do this, download the plugin here, add it in a script tag in the HTML page after Backbone and initialize it in the collection like this:

```
var Library = Backbone.Collection.extend({
    localStorage:new Backbone.LocalStorage("LibraryStorage"),
    model:Book,
    url:'/api/books'
});
```

By this Backbone will assume that the API looks like this:

```
url HTTP Method Operation
/api/books  GET Get an array of all books
/api/books/:id  GET Get the book with id of :id
/api/books  POST  Add a new book and return the book with an id attribute added
/api/books/:id  PUT Update the book with id of :id
/api/books/:id  DELETE  Delete the book with id of :id
```

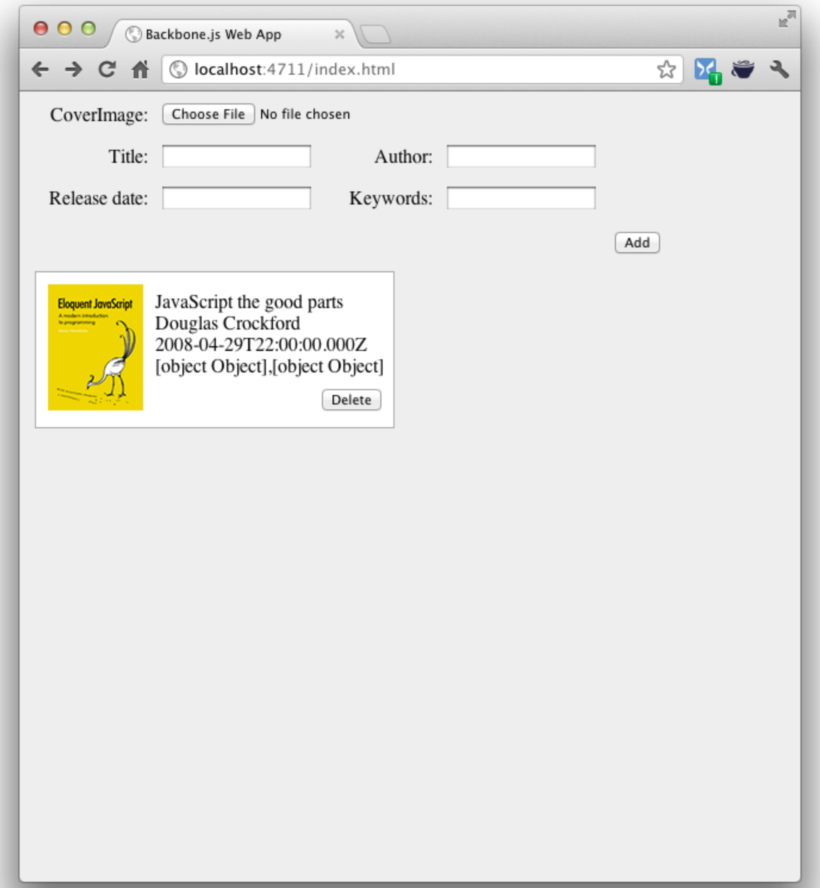
To make our application get the Book models from the server on page load we need to update the LibraryView. It is recommended in the Backbone documentation to insert all models when the page is generated on the server side, rather than fetching them from the client side once the page is loaded. Since this tutorial will try to give you a more complete picture of how to communicate with a server, we will go ahead and ignore that recommendation. Go to the LibraryView declaration in app.js and update the initialize function as follows:

```
initialize:function () {
    //this.collection = new Library(books);
    this.collection = new Library();
    this.collection.fetch();
    this.render();

    this.collection.on("add", this.renderBook, this);
    this.collection.on("remove", this.removeBook, this);
    this.collection.on("reset", this.render, this);
},
```

Here I have replaced the row where we create a collection from the internal array with `this.collection.fetch()`. I have also added a listener on the reset event. We need to do this since the fetching of models is asynchronous and happens after the page is rendered. When the fetching is finished, Backbone will fire the reset event, which we listen to and re-render the view. If you reload the page now you should see all books that are stored on the server:

As you can see the date and keywords look a bit weird. The date delivered from the server is converted into a JavaScript Date object and when applied to the underscore template it will use the toString() function to display it. There isn't very good support for formatting dates in JavaScript so we will use the dateFormat jQuery plugin to fix this. Go ahead and download it from here and put it in your js folder. Then go ahead and change index.html like this:

```
<body>
<div id="books">
    <form id="addBook" action="#">
        <div>
            <label for="coverImage">CoverImage: </label><input id="coverImage" type="file">
            <label for="title">Title: </label><input id="title">
            <label for="author">Author: </label><input id="author">
            <label for="releaseDate">Release date: </label><input id="releaseDate">
            <label for="keywords">Keywords: </label><input id="keywords">
            <button id="add">Add</button>
        </div>
    </form>
</div>
<script id="bookTemplate" type="text/template">
    <img src="<%= coverImage %>"/>
    <ul>
        <li><%= title%></li>
        <li><%= author%></li>
        <li><%= $.format.date(new Date(releaseDate), 'dd/MM/yyyy') %></li>
        <li><%= keywords %></li>
    </ul>
    <button class="delete">Delete</button>
</script>
```



CoverImage: No file chosen

Title: Author:

Release date: Keywords:

JavaScript the good parts
Douglas Crockford
2008-04-29T22:00:00.000Z
[object Object],[object Object]

```
<script src="js/jquery-1.7.1.js"></script>
<script src="js/jquery.dateFormat-1.0.js"></script>
<script src="js/underscore.js"></script>
<script src="js/backbone.js"></script>
<script src="js/app.js"></script>
</body>
```

So I added the jquery.dateFormat-1.0.js file on row 25 and used dateFormat to print the date in dd/MM/yyyy format on line 19. Now the date on the page should look a bit better. How about the keywords? Since we are receiving the keywords in an array we need to execute some code that generates a string of separated keywords. To do that we can omit the equals letter in the template tag which will let us execute code that doesn't display anything:

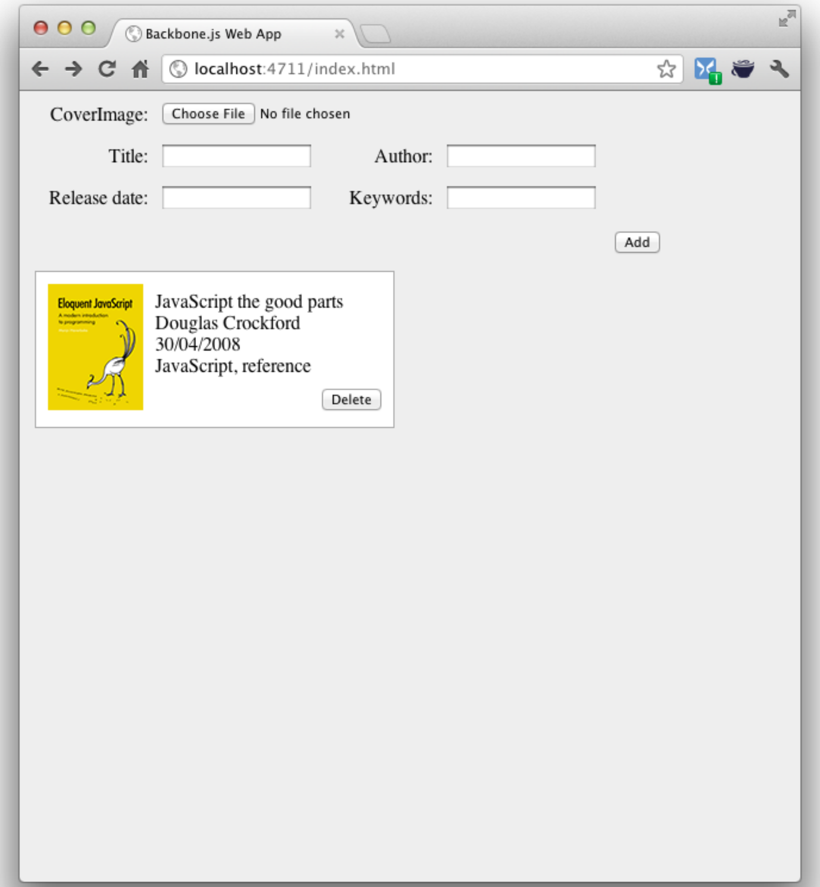
```
<script id="bookTemplate" type="text/template">
    <img src="<%= coverImage %>"/>
    <ul>
        <li><%= title%></li>
        <li><%= author%></li>
        <li><%= $.format.date(new Date(releaseDate), 'dd/MM/yyyy') %></li>
        <li><% _.each(keywords, function(keyobj){%> <%= keyobj.keyword %><% }); %></li>
    </ul>
    <button class="delete">Delete</button>
</script>
```

Here I iterate over the keywords array using the each function and print out every single keyword. Note that I display the keyword using the <%= tag. This will display the keywords with a space between. If you would like to support space characters within a keyword, like for example "new age" then you could separate them by comma (no sane person would put a comma in a keyword right?). That would lead us to write some more code to remove the comma after the last keyword like this:

```
<li>
    <% _.each(keywords, function(keyobj, index, list){ %>
        <%= keyobj.keyword %>
        <% if(index < list.length - 1){ %>
            <%= ', ' %>
        <% } %>
    <% }); %>
</li>
```

Reloading the page again should look quite decent:

Now go ahead and delete a book and then reload the page: Tadaa! the deleted book is back! Not cool, why is this? This happens because when we get the



JavaScript the good parts
Douglas Crockford
30/04/2008
JavaScript, reference

BookModels from the server they have an _id attribute (notice the underscore), but Backbone expects an id attribute (no underscore). Since no id attribute is present, Backbone sees this model as new and deleting a new model don't need any synchronization. To fix this we could change the server response, but we are instead going to look at the parse function of Backbone.Model. The parse function lets you edit the server response before it is passed to the Model constructor. Update the Book model like this:

```
var Book = Backbone.Model.extend({
    defaults:{
        coverImage:"img/placeholder.png",
        title:"No title",
        author:"Unknown",
        releaseDate:"Unknown",
        keywords:"None"
    },
    parse:function (response) {
        console.log(response);
        response.id = response._id;
        return response;
    }
});
```

I simply copy the value of _id to the needed id attribute. If you reload the page you will see that models are actually deleted on the server when you press the delete button.

Note: A simpler way of making Backbone recognize _id as its unique identifier is to set the idAttribute of the model like this:

```
var Book = Backbone.Model.extend({
    defaults:{
        coverImage:"img/placeholder.png",
        title:"No title",
        author:"Unknown",
        releaseDate:"Unknown",
        keywords:"None"
    },
    idAttribute:"_id"
});
```

If you now try to add a new book using the form you'll notice that it is the same story as with delete – models wont get persisted on the server. This is because Backbone.Collection.add doesn't automatically sync, but it is easy to fix. In LibraryView in app.js, change the line reading:

```
this.collection.add(new Book(formData));
```

. . . to:

```
this.collection.create(formData);
```

Now newly created books will get persisted. Actually, they probably wont if you enter a date. The server expects a date in UNIX timestamp format (milliseconds since Jan 1, 1970). Also any keywords you enter wont be stored since the server expects an array of objects with the attribute 'keyword'.

We start with fixing the date issue. We don't really want the users to enter a date into a specific format manually so we'll use the standard datepicker from jQuery UI. Go ahead and create a custom jQuery UI download containing datepicker from here. Add the css and js files to index.html:

```
<head>
    <meta charset="UTF-8"/>
    <title>Backbone.js Web App</title>
    <link rel="stylesheet" href="css/screen.css">
    <link rel="stylesheet" href="css/cupertino/jquery-ui-1.8.19.custom.css">
</head>
```

and the js file after jQuery:

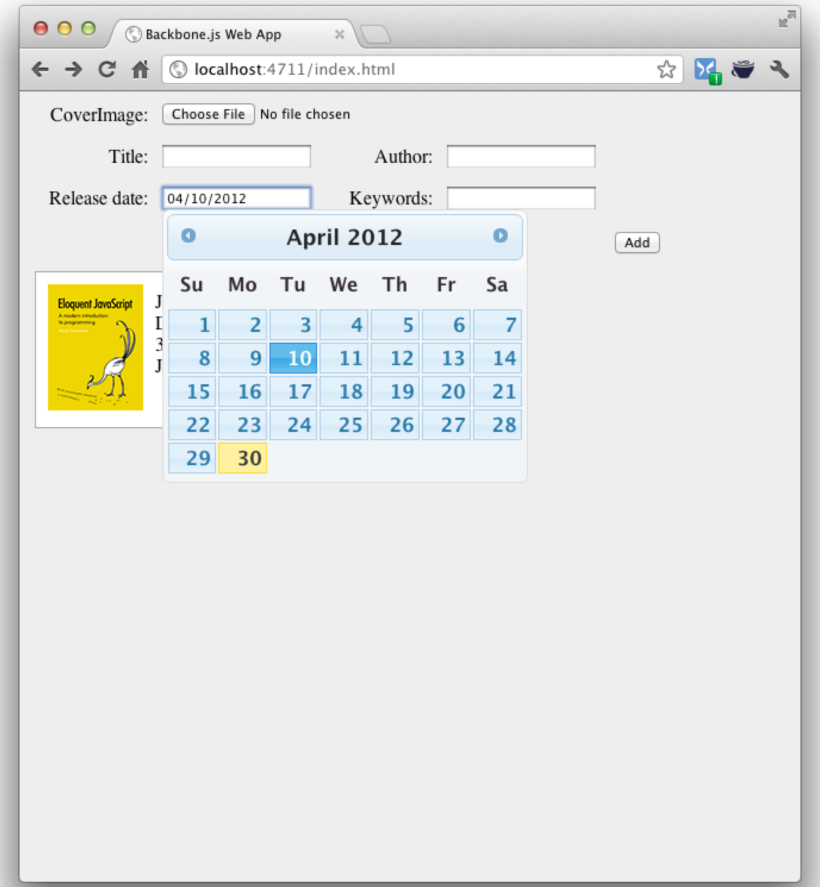
```
<script src="js/jquery-1.7.1.js"></script>
<script src="js/jquery-ui-1.8.19.custom.min.js"></script>
<script src="js/jquery.dateFormat-1.0.js"></script>
<script src="js/underscore.js"></script>
<script src="js/backbone.js"></script>
<script src="js/app.js"></script>
```

Now in the beginning of app.js bind datepicker to our releaseDate field:

```
(function ($) {
    $("#releaseDate").datepicker();

    var books = [
        {title:"JS the good parts", author:"John Doe", releaseDate:"2012", keywords:"JavaScr
```

You should now be able to pick a date when clicking in the releaseDate field:

Now go to the addBook function in LibraryView and update it like this:


Backbone.js Web App
localhost:4711/index.html
CoverImage: Choose File No file chosen
Title:
Author:
Release date: 04/10/2012
Keywords:
Add
April 2012
Su Mo Tu We Th Fr Sa
1 2 3 4 5 6 7
8 9 10 11 12 13 14
15 16 17 18 19 20 21
22 23 24 25 26 27 28
29 30

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

    var formData = {};

    $("#addBook div").children("input").each(function (i, el) {
        if ($(el).val() !== "") {
            if (el.id === 'releaseDate'){
                formData[el.id] = $('#releaseDate').datepicker("getDate").getTime();
            } else {
                formData[el.id] = $(el).val();
            }
        }
    });

    books.push(formData);

    //this.collection.add(new Book(formData));
    this.collection.create(formData);
},
```

Here I check if the current element is the releaseDate input field, in which case I use datePicker("getDate") which will give me a Date object and then use the getTime function on that to get the time in milliseconds. While we are at it lets fix the keywords issue as well:

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

    var formData = {};

    $("#addBook div").children("input").each(function (i, el) {
        if ($(el).val() !== "") {
            if (el.id === 'keywords') {
                var keywordArray = $(el).val().split(',');
                var keywordObjects = [];
                for (var j = 0; j < keywordArray.length; j++) {
                    keywordObjects[j] = {"keyword":keywordArray[j]};
                }
                formData[el.id] = keywordObjects;
            } else if (el.id === 'releaseDate'){
                formData[el.id] = $('#releaseDate').datepicker("getDate").getTime();
            } else {
                formData[el.id] = $(el).val();
            }
        }
    });

    books.push(formData);

    //this.collection.add(new Book(formData));
    this.collection.create(formData);
},
```

Here I check if the current element is the keywords input field, in which case I split the string on each comma and then create a new array with keyword objects. In other words I assume that keywords are separated by commas, so I better write a comment on that in the form. Now you should be able to add new books with both release date and keywords!

Connecting with a third party API

In a real world scenario, the model and the view can be adapted to connect a third party API. For example:

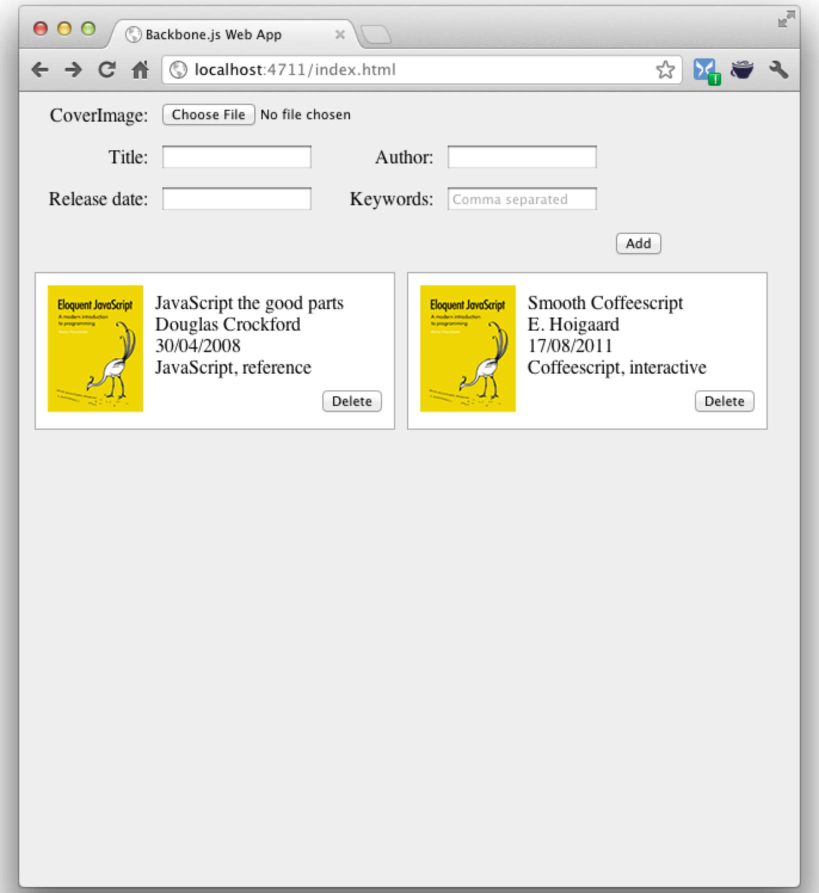
```
url HTTP Method Operation
index.php?option=com_todo&view=books&task=get  GET Get an array of all books
```

In the model, you can define the third party API URL via url definition.

```
url : function() {
  return 'index.php?option=com_todo&view=books&task=get';
}
```

In the view, you can define the attributes to be processed by the model.

```
// Third party API example.
this.collection.create(this.collection.model, {
    attrs : {
       bookId: this.$('#book_id').val()
    },

    // Wait for the answer.
    wait : true,

    // Report if there's any error.
    error : function(model, fail, xhr) {
        ....
    }
});
```



JavaScript the good parts
Douglas Crockford
30/04/2008
JavaScript, reference

Smooth Coffeescript
E. Hoigaard
17/08/2011
Coffeescript, interactive

Summary

In this tutorial we made our application persistent by binding it to a server using a REST API. We also looked at some problems that might occur when serializing and deserializing data and their solutions. We looked at the dateFormat and the datepicker jQuery plugins and how to do some more advanced things in our Underscore templates. The code is available here.

Backbone Boilerplate And Grunt-BBB

Backbone Boilerplate is an excellent set of best practices and utilities for building Backbone.js applications, created by Backbone contributor Tim Branyen. He organized this boilerplate out of the gotchas, pitfalls and common tasks he ran into over a year of heavily using Backbone to build apps at Bocoup. This includes apps such StartupDataTrends.com.

With scaffolding and built in build tasks that take care of minification, concatentation, server, template compilation and more, Backbone Boilerplate (and sister project Grunt-BBB) are an excellent choice for developers of all levels. I heavily recommend using them as they will give you an enormous start when it comes to getting setup for development. They also have some great inline documentation which is also another excellent time-saver.

By default, Backbone Boilerplate provides you with:

- Backbone, Lodash (an Underscore.js alternative) and jQuery with an HTML5 Boilerplate foundation
- Boilerplate module code
- A Windows/Mac/Linux build tool for template precompilation and, concatenation & minification of all your libraries, application code and CSS
- Scaffolding support (via grunt-bbb - [B]ackbone [B]oilerplate [B]uild) so you have to spend minimal time writing boilerplate for modules, collections and so on.
- A Lightweight node.js webserver
- Numerous other Backbone.js snippets for making your life easier

Getting Started

Backbone Boilerplate

We can use Boilerplate to easily begin creating an application, but first, we'll need to install it. This can be done by grabbing the latest version of it by cloning the Boilerplate repo directly:

```
$ git clone git://github.com/tbranyen/backbone-boilerplate.git
```

or alternatively, just fetching the latest tarball as follows:

```
$ curl -C - -O https://github.com/tbranyen/backbone-boilerplate/zipball/master
```

Grunt-BBB

As Tim covers in the Boilerplate docs, we have to install Grunt if we want to use the build tools and grunt-bbb helpers he recommends.

Grunt is an excellent Node-based JavaScript build tool by another Bocoup developer (Ben Alman). Think of it as similar to Ant or Rake. The grunt-bbb helper is also useful to have as it provides several Backbone-specific utilities for scaffolding out your project, without the need to write boilerplate yourself.

To install grunt and grunt-bbb via NPM:

```
# first run
$ npm install -g grunt

# followed by
$ npm install -g bbb

# finally create a new project
$ bbb init
```

That's it. We should now be good to go.

A typical workflow for using grunt-bbb, which we can use later on is:

- Initialize a new project (`bbb init`)
- Add new modules and templates (`bbb init:module`)
- Develop using the built in server (`bbb server`)
- Run the build tool (`bbb build`)
- Deploy and map to production assets (using `bbb release`)

Creating a new project

Let's create a new folder for our project and run `bbb init` to kick things off. If everything has been correctly installed, this will sub out some project directories and files for us. Let's review what is generated.

index.html

This is a fairly standard stripped-down HTML5 Boilerplate foundation with the notable exception of including RequireJS at the bottom of the page.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge,chrome=1">
  <meta name="viewport" content="width=device-width,initial-scale=1">

  <title>Backbone Boilerplate</title>

  <!-- Application styles. -->
  <link rel="stylesheet" href="/assets/css/index.css">
</head>

<body>
  <!-- Main container. -->
  <div role="main" id="main"></div>

  <!-- Application source. -->
  <script data-main="/app/config" src="/assets/js/libs/require.js"></script>
</body>
</html>
```

RequireJS is an AMD (Asynchronous Module Definition) module and script loader, which will assist us with managing the modules in our application. We'll be covering it in a lot more detail later on in the book, but for now, let's cover at a high-level what this particular block does:

```
<script data-main="app/config" src="/assets/js/libs/require.js"></script>
```

The `data-main` attribute is used to inform RequireJS to load `app/config.js` (a configuration object) after it has finished loading itself. You'll notice that we've omitted the `.js` extension here as RequireJS can automatically add this for us, however it will respect your paths if we do choose to include it regardless. Let's now look at the config file being referenced.

config.js

A RequireJS configuration object allows us to specify aliases and paths for dependencies we're likely to reference often (e.g jQuery), bootstrap properties like our base application URL and `shim` libraries that don't support AMD natively.

This is what the config file in Backbone Boilerplate looks like:

```
// Set the RequireJS configuration for your application.
require.config({

  // Initialize the application with the main application file.
  deps: ['main'],

  paths: {
    // JavaScript folders.
    libs: '../assets/js/libs',
    plugins: '../assets/js/plugins',

    // Libraries.
    jquery: '../assets/js/libs/jquery',
    lodash: '../assets/js/libs/lodash',
    backbone: '../assets/js/libs/backbone'
  },

  shim: {
    // Backbone library depends on lodash and jQuery.
    backbone: {
      deps: ['lodash', 'jquery'],
      exports: 'Backbone'
    },

    // Backbone.LayoutManager depends on Backbone.
    'plugins/backbone.layoutmanager': ['backbone']
  }

});
```

The first option defined in the above config is `deps: ['main']`. This informs RequireJS to load up our main.js file, which is considered the entry point for our application. You may notice that we haven't specified any other path information for `main`.

This is because as we haven't overridden the path to our scripts using the `baseUrl` option, Require will infer this using the path from our `data-main` attribute in index.html. In other words, our `baseUrl` is `app/` and any scripts we require will be loaded relative to this location.

The next block is `paths`, which we can use to specify paths relative to the `baseUrl` as well as the paths/aliases to dependencies we're likely to regularly reference.

```
  paths: {
    // JavaScript folders.
    libs: '../assets/js/libs',
    plugins: '../assets/js/plugins',

    // Libraries.
    jquery: '../assets/js/libs/jquery',
    lodash: '../assets/js/libs/lodash',
    backbone: '../assets/js/libs/backbone'
  },
```

Next we have the `shim` config:

```
  shim: {
    // Backbone library depends on lodash and jQuery.
    backbone: {
      deps: ['lodash', 'jquery'],
      exports: 'Backbone'
    },

    // Backbone.LayoutManager depends on Backbone.
    'plugins/backbone.layoutmanager': ['backbone']
  }
```

`shim` is an important part of our RequireJS configuration which allows us to load libraries which are not AMD compliant. The basic idea here is that rather than requiring all libraries to implement support for AMD, the `shim` takes care of the hard work for us.

For example, in the block below, we state that Backbone.js is dependent on Lodash (a fork of Underscore.js) and jQuery being loaded before it. Once they've been loaded, we then use the global export `Backbone` as the module value.

```
    backbone: {
      deps: ['lodash', 'jquery'],
      exports: 'Backbone'
    }
```

Finally, we inform RequireJS that the Backbone LayoutManager plugin (a template and layout manager, also included) requires that Backbone be loaded before it should be.

```
    // Backbone.LayoutManager depends on Backbone.
    'plugins/backbone.layoutmanager': ['backbone']
```

This entire setup ensures that our scripts correctly get loaded in the order in which we expect.

main.js

Next, we have `main.js`, which defines the entry point for our application. We use a global `require()` method to load an array any other scripts needed, such as our application `app.js` and our main router `router.js`. Note that most of the time, we will only use `require()` for bootstrapping an application and a similar method called `define()` for all other purposes.

The function defined after our array of dependencies is a callback which doesn't fire until these scripts have loaded. Notice how we're able to locally alias references to "app" and "router" as `app` and `Router` for convenience.

```
require([
  // Application.
  'app',

  // Main Router.
  'router'
],

function(app, Router) {

  // Define your master router on the application namespace and trigger all
  // navigation from this instance.
  app.router = new Router();

  // Trigger the initial route and enable HTML5 History API support, set the
  // root folder to '/' by default.  Change in app.js.
  Backbone.history.start({ pushState: true, root: app.root });

  // All navigation that is relative should be passed through the navigate
  // method, to be processed by the router. If the link has a `data-bypass`
  // attribute, bypass the delegation completely.
  $(document).on('click', 'a:not([data-bypass])', function(evt) {
    // Get the absolute anchor href.
    var href = $(this).attr('href');

    // If the href exists and is a hash route, run it through Backbone.
    if (href && href.indexOf('#') === 0) {
      // Stop the default event to ensure the link will not cause a page
      // refresh.
      evt.preventDefault();

      // `Backbone.history.navigate` is sufficient for all Routers and will
      // trigger the correct events. The Router's internal `navigate` method
      // calls this anyways.  The fragment is sliced from the root.
      Backbone.history.navigate(href, true);
    }
  });

});
```

Inline, Backbone Boilerplate includes boilerplate code for initializing our router with HTML5 History API support and handling other navigation scenarios, so we don't have to.

app.js

Let us now look at our `app.js` module. Typically, in non-Backbone Boilerplate applications, an `app.js` file may contain the core logic or module references needed to kick start an app.

In this case however, this file is used to define templating and layout configuration options as well as utilities for consuming layouts. To a beginner, this might look like a lot of code to comprehend, but the good news is that for basic apps, you're unlikely to need to heavily modify this. Instead, you'll be more concerned with modules for your app, which we'll look at next.

```
define([

  // Libraries.
  'jquery',
  'lodash',
  'backbone',

  // Plugins.
  'plugins/backbone.layoutmanager'

],

function($, _, Backbone) {

  // Provide a global location to place configuration settings and module
  // creation.
  var app = {
    // The root path to run the application.
    root: '/'
  };

  // Localize or create a new JavaScript Template object.
  var JST = window.JST = window.JST || {};
```

```
// Configure LayoutManager with Backbone Boilerplate defaults.
Backbone.LayoutManager.configure({
  paths: {
    layout: 'app/templates/layouts/',
    template: 'app/templates/'
  },

  fetch: function(path) {
    path = path + '.html';

    if (!JST[path]) {
      $.ajax({ url: app.root + path, async: false }).then(function(contents) {
        JST[path] = _.template(contents);
      });
    }

    return JST[path];
  }
});

// Mix Backbone.Events, modules, and layout management into the app object.
return _.extend(app, {
  // Create a custom object with a nested Views object.
  module: function(additionalProps) {
    return _.extend({ Views: {} }, additionalProps);
  },

  // Helper for using layouts.
  useLayout: function(name) {
    // If already using this Layout, then don't re-inject into the DOM.
    if (this.layout && this.layout.options.template === name) {
      return this.layout;
    }

    // If a layout already exists, remove it from the DOM.
    if (this.layout) {
      this.layout.remove();
    }

    // Create a new Layout.
    var layout = new Backbone.Layout({
      template: name,
      className: 'layout ' + name,
      id: 'layout'
    });
```

```
      // Insert into the DOM.
      $('#main').empty().append(layout.el);

      // Render the layout.
      layout.render();

      // Cache the reference.
      this.layout = layout;

      // Return the reference, for chainability.
      return layout;
    }
  }, Backbone.Events);

});
```

Creating Backbone Boilerplate Modules

Not to be confused with simply being just an AMD module, a Backbone Boilerplate `module` is a script composed of a:

- Model
- Collection
- Views (optional)

We can easily create a new Boilerplate module using `grunt-bbb` once again using `init`:

```
# Create a new module
$ bbb init:module

# Grunt prompt
Please answer the following:
[?] Module Name foo
[?] Do you need to make any changes to the above before continuing? (y/N) n

Writing app/modules/foo.js...OK

Initialized from template "module".
```

This will generate a module `foo.js` as follows:

```
define([
  // Application.
  'app'
],

// Map dependencies from above array.
function(app) {

  // Create a new module.
  var Foo = app.module();

  // Default model.
  Foo.Model = Backbone.Model.extend({

  });

  // Default collection.
  Foo.Collection = Backbone.Collection.extend({
    model: Foo.Model
  });

  // Return the module for AMD compliance.
  return Foo;

});
```

Notice how boilerplate code for our model and collection has already been written for us, as well as code for consuming the layout utilities defined in `app.js`.

Now, you may be wondering where or how Views fit into this setup. Although Backbone Boilerplate doesn't include Views in its generated modules by default, we can easily add them ourselves as needed.

e.g:

```
define([
  // Application.
  'app',

  // Views
  'modules/foo/views'
],

// Map dependencies from above array.
function(app, Views) {
  // Create a new module.
  var Foo = app.module();

  // Default model.
  Foo.Model = Backbone.Model.extend({

  });

  // Default collection.
  Foo.Collection = Backbone.Collection.extend({
    model: Foo.Model
  });

  // Default views
  Foo.Views = Views;

  // Return the module for AMD compliance.
  return Foo;

});
```

Optionally, we may also wish to include references to plugins such as the Backbone LocalStorage or Offline adapters. One clean way of including a plugin in the above boilerplate could be:

```
define([
  'app',

  // Libs
  'backbone',

  // Views
  'modules/foo/views',

  // Plugins
  'plugins/backbone-localstorage'
],

function(app, Backbone, Views) {
  // Create a new module.
  var Foo = app.module();

  // Default model.
  Foo.Model = Backbone.Model.extend({
  });

  // Default collection.
  Foo.Collection = Backbone.Collection.extend({
    model: Foo.Model,

    // Save all of the items under the `"foo"` namespace.
    localStorage: new Store('foo-backbone'),
  });

  // Default views
  Foo.Views = Views;

  // Return the module for AMD compliance.
  return Foo;
});
```

You may have spotted that in our module sample we're using the plural, "Views", rather than just View. This is because a View module can contain references to as many Views as needed. In the above, our `/modules/foo/views.js` file may look as follows:

```
define([
  'app',

  // Libs
  'backbone'
],

function(app, Backbone) {

  var Views = {};

  Views.Bar = Backbone.View.extend({
    template: 'foo/bar',
    tagName: 'li',
    ...
   });

  Views.Baz = Backbone.View.extend({
    template: 'foo/baz',
    tagName: 'li',
    ...
   });
   return Views;

});
```

Where the `template` references in our Views, correspond to files in the `app/templates` directory. e.g `foo/bar` is located at `app/templates/foo/bar.html` and is a HTML template that can contain Lodash/Underscore.js Micro-templating logic.

router.js

Finally, let's look at our application router, used for handling navigation. The default router Backbone Boilerplate generates for us inclues sane defaults for no routes being specified.

```
define([
  // Application.
  'app'
],

function(app) {

  // Defining the application router, you can attach sub routers here.
  var Router = Backbone.Router.extend({
    routes: {
      '': 'index'
    },

    index: function() {

    }
  });

  return Router;

});
```

If however we would like to execute some module-specific logic, when the page loads (i.e when a user hits the default route), we can pull in a module as a dependency and optionally use the Backbone LayoutManager to attach Views to our layout as follows:

```
define([
  // Application.
  'app',

  // Modules
  'modules/foo'
],

function(app, Foo) {

  // Defining the application router, you can attach sub routers here.
  var Router = Backbone.Router.extend({
    routes: {
      '': 'index'
    },

    index: function() {
            // Create a new Collection
            var collection = new Foo.Collection();

            // Use and configure a 'main' layout
            app.useLayout('main').setViews({
                    // Attach the bar View into the content View
                    '.bar': new Foo.Views.Bar({
                            collection: collection
                    })
             }).render();
    }
  });

  // Fetch data (e.g from localStorage)
  collection.fetch();

  return Router;

});
```

Conclusions

In this section we reviewed Backbone Boilerplate and learned how to use the BBB tool to help us scaffold out our application.

If you would like to learn more about how this project helps structure your app, BBB includes some built-in boilerplate sample apps that can be easily generated for review.

These include a boilerplate tutorial project (`bbb init:tutorial`) and an implementation of my TodoMVC project (`bbb init:todomvc`). I recommend checking these out as they'll provide you with a more complete picture of how Backbone Boilerplate, its templates and so on fit into the overall setup for a web app.

For more about Grunt-BBB, remember to take a look at the official project repository. There is also a related slide-deck available for those interested in reading more.

Related Tools & Projects

As we've seen, scaffolding tools can assist in expediting how quickly you can begin a new application by creating the basic files required for a project automatically. If you appreciate such tools, I'm happy to also recommend checking out Yeoman (one of my upcoming projects) and Brunch.

Brunch works very well with Backbone, Underscore, jQuery and CoffeeScript and is even used by companies such as Red Bull and Jim Beam. You may have to update any third party dependencies (e.g. latest jQuery or Zepto) when using it, but other than that it should be fairly stable to use right out of the box.

Brunch can be installed via the nodejs package manager and is easy to get started with. If you happen to use Vim or Textmate as your editor of choice, you'll be happy to know that there are Brunch bundles available for both.

Common Problems & Solutions

In this section, we will review a number of common problems developers often experience once they've started to work on relatively non-trivial projects using Backbone.js, as well as present potential solutions.

Perhaps the most frequent of these questions surround how to do more with Views. If you are interested in discovering how to work with nested Views, learn about view disposal and inheritance, this section will hopefully have you covered.

Nesting: What is the best approach for rendering and appending Sub-Views in Backbone.js? Nesting is generally considered a good way to maintain hierarchal views for writing maintainable code. As a beginner, one might try writing a very simple setup with sub-views (e.g inner views) as follows:

```
// Where we have previously defined a View, SubView
// in a parent View we could do:

...
initialize : function () {
    this.innerView1 = new Subview({options});
    this.innerView2 = new Subview({options});
},

render : function () {

    this.$el.html(this.template());

    this.innerView1.setElement('.some-element').render();
    this.innerView2.setElement('.some-element').render();
}
```

This works in that one doesn't need to worry about maintaining the order of your DOM elements when appending. Views are initialized early and the render() method doesn't need to take on too many responsibilities at once. Unfortunately, a downside is that you don't have the ability to set the `tagName` of elements and events need to be re-delegated.

An alternative approach which doesn't suffer from the re-delegation problem could be written as follows:

```
initialize : function () {

},

render : function () {

    this.$el.empty();

    this.innerView1 = new Subview({options});
    this.innerView2 = new Subview({options});


    this.$el.append(this.innerView1.render().el, this.innerView2.render().el);
}
```

In this version, we also don't require a template containing empty placeholders and the issue with `tagNames` is solved as they are defined by the view once again.

Yet another variation which moves logic into an `onRender` event, could be written with only a few subtle changes:

```
initialize : function () {
    this.on('render', this.onRender);
},

render : function () {

    this.$el.html(this.template);

    // more logic

    return this.trigger('render');
},

onRender : function () {
    this.innerView1 = new Subview();
    this.innerView2 = new Subview();
    this.innerView1.setElement('.some-element').render();
    this.innerView2.setElement('.some-element').render();
}
```

If you find yourself nesting views in your application, there are more optimal approaches possible for initializing, rendering and appending your sub-views. One such solution could be written:

```
var OuterView = Backbone.View.extend({
    initialize: function() {
        this.inner = new InnerView();
    },

    render: function() {
        this.$el.html(template); // or this.$el.empty() if you have no template
        this.$el.append(this.inner.$el);
        this.inner.render();
    }
});

var InnerView = Backbone.View.extend({
    render: function() {
        this.$el.html(template);
        this.delegateEvents();
    }
});
```

This tackles a few specific design decisions:

- The order in which you append the sub-elements matters

- The OuterView doesn't contain the HTML elements to be set in the InnerView(s), meaning that we can still specify tagName in the InnerView
- render() is called after the InnerView element has been placed into the DOM. This is useful if your InnerView's render() method is sizing itself on the page based on the dimensions of another element. This is a common use case.

A second potential solution is this, which may appear cleaner but in reality has a tendency to affect performance:

```
var OuterView = Backbone.View.extend({
    initialize: function() {
        this.render();
    },

    render: function() {
        this.$el.html(template); // or this.$el.empty() if you have no template
        this.inner = new InnerView();
        this.$el.append(this.inner.$el);
    }
});

var InnerView = Backbone.View.extend({
    initialize: function() {
        this.render();
    },

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

If multiple views need to be nested at particular locations in a template, a hash of child views indexed by child view cids' should be created. In the template, use a custom HTML attribute named `data-view-cid` to create placeholder elements for each view to embed. Once the template has been rendered and it's output appeneded to the parent view's `$el`, each placeholder can be queried for and replaced with the child view's `el`.

A sample implementation containing a single child view:

```
var OuterView = Backbone.View.extend({
    initialize: function() {
        this.children = {};
        var child = new Backbone.View();
        this.children[child.cid] = child;
    },

    render: function() {
        this.$el.html('<div data-view-cid="' + this.child.cid + '"></div>');
        _.each(this.children, function(view, cid) {
            this.$('[data-view-cid="' + cid + '"]').replaceWith(view.el);
        }, this);
    }
};
```

Generally speaking, more developers opt for the first solution as:

- The majority of their views may already rely on being in the DOM in their render() method
- When the OuterView is re-rendered, views don't have to be re-initialized where re-initialization has the potential to cause memory leaks and issues with existing bindings

Marionette and Thorax provide logic for nesting views, and rendering collections where each item has an associated view. Marionette provides APIs in JavaScript while Thorax provides APIs via Handlebars template helpers.

(Thanks to Lukas and Ian Taylor for these tips).

What is the best way to manage models in nested Views? In order to reach attributes on related models in a nested setup, the models involved need to have some prior knowledge about which models this refers to. Backbone.js doesn't implicitly handle relations or nesting, meaning it's up to us to ensure models have a knowledge of each other.

One approach is to make sure each child model has a 'parent' attribute. This way you can traverse the nesting first up to the parent and then down to any siblings that you know of. So, assuming we have models modelA, modelB and modelC:

```
// When initializing modelA, I would suggest setting a link to the parent
// model when doing this, like this:

ModelA = Backbone.Model.extend({

    initialize: function(){
        this.modelB = new modelB();
        this.modelB.parent = this;
        this.modelC = new modelC();
        this.modelC.parent = this;
    }
}
```

This allows you to reach the parent model in any child model function by calling this.parent.

When you have a need to nest Backbone.js views, you might find it easier to let each view represent a single HTML tag using the tagName option of the View. This may be written as:

```
ViewA = Backbone.View.extend({

    tagName: 'div',
    id: 'new',

    initialize: function(){
       this.viewB = new ViewB();
       this.viewB.parentView = this;
       $(this.el).append(this.viewB.el);
    }
});

ViewB = Backbone.View.extend({

    tagName: 'h1',

    render: function(){
        $(this.el).html('Header text'); // or use this.options.headerText or equivalent
    },

    funcB1: function(){
        this.model.parent.doSomethingOnParent();
        this.model.parent.modelC.doSomethingOnSibling();
        $(this.parentView.el).shakeViolently();
    }

});
```

Then in your application initialization code , you would initiate ViewA and place its element inside the body element.

An alternative approach is to use an extension called Backbone-Forms. Using a similar schema to what we wrote earlier, nesting could be achieved as follows:

```
var ModelB = Backbone.Model.extend({
    schema: {
        attributeB1: 'Text',
        attributeB2: 'Text'
    }
});

var ModelC = Backbone.Model.extend({
    schema: {
        attributeC: 'Text',
    }
});

var ModelA = Backbone.Model.extend({
    schema: {
        attributeA1: 'Text',
        attributeA2: 'Text',
        refToModelB: { type: 'NestedModel', model: ModelB, template: 'templateB' },
        refToModelC: { type: 'NestedModel', model: ModelC, template: 'templateC' }
    }
});
```

There is more information about this technique available on GitHub.

(Thanks to Jens Alm and Artem Oboturov for these tips)

Is it possible to have one Backbone.js View trigger updates in other Views? The Mediator pattern is an excellent option for implementing a solution to this problem.

Without going into too much detail about the pattern, it can effectively be used an event manager that lets you to subscribe to and publish events. So an ApplicationViewA could subscribe to an event, i.e. 'selected' and then the ApplicationViewB would publish the 'selected' event.

The reason I like this is it allows you to send events between views, without the views being directly bound together.

For Example:

```
// See http://addyosmani.com/largescalejavascript/#mediatorpattern
// for an implementation or alternatively for a more thorough one
// http://thejacklawson.com/Mediator.js/

var mediator = new Mediator();
var ApplicationViewB = Backbone.View.extend({
    toggle_select: function() {
        ...
        mediator.publish('selected', any, data, you, want);
        return this;
    }
});

var ApplicationViewA = Backbone.View.extend({
    initialize: function() {
        mediator.subscribe('selected', this.delete_selected)
    },

    delete_selected: function(any, data, you, want) {
        ... do something ...
    },
});
```

This way your ApplicationViewA doesn't care if it is an ApplicationViewB or FooView that publishes the 'selected' event, only that the event occurred. As a result, you may find it a maintainable way to manage events between parts of your application, not just views.

(Thanks to John McKim for this tip and for referencing my Large Scale JavaScript Patterns article).

How would one render a Parent View from one of its Children? If you say, have a view which contains another view (e.g a main view containing a modal view) and would like to render or re-render the parent view from the child, this is extremely straight-forward.

In such a scenario, you would most likely want to execute the rendering when a particular event has occurred. For the sake of example, let us call this event 'somethingHappened'. The parent view can bind notifications on the child view to know when the event has occurred. It can then render itself.

On the parent view:

```
// Parent initialize
this.childView.on('somethingHappened', this.render, this);

// Parent removal
this.childView.off('somethingHappened', this.render, this);
```

On the child view:

```
// After the event has occurred
this.trigger('somethingHappened');
```

The child will trigger a "somethingHappened" event and the parent's render function will be called.

(Thanks to Tal Bereznitskey for this tip)

How do you cleanly dispose Views to avoid memory leaks? As your application grows, keeping live views around which aren't being used can quickly become difficult to maintain. Instead, you may find it more optimal to destroy views that are no longer required and simply create new ones as the necessity arises.

A solution to help with this is to create a BaseView from which the rest of your views inherit from. The idea here is that your view will maintain a reference to all of the events to which its subscribed to so that when it is time to dispose of a view, all of those bindings will be automatically unbound.

Here is a sample implementation of this:

```
var BaseView = function (options) {

    this.bindings = [];
    Backbone.View.apply(this, [options]);
};

_.extend(BaseView.prototype, Backbone.View.prototype, {

    bindTo: function (model, ev, callback) {

        model.bind(ev, callback, this);
        this.bindings.push({ model: model, ev: ev, callback: callback });
    },

    unbindFromAll: function () {
        _.each(this.bindings, function (binding) {
            binding.model.unbind(binding.ev, binding.callback);
        });
        this.bindings = [];
    },

    dispose: function () {
        this.unbindFromAll(); // this will unbind all events that this view has bound to
        this.unbind(); // this will unbind all listeners to events from this view. This is
        this.remove(); // uses the default Backbone.View.remove() method which removes this
    }

});

BaseView.extend = Backbone.View.extend;
```

Then, whenever a view has the need to bind to an event on a model or a collection, you would use the bindTo method. e.g:

```
var SampleView = BaseView.extend({

    initialize: function(){
        this.bindTo(this.model, 'change', this.render);
        this.bindTo(this.collection, 'reset', this.doSomething);
    }
});
```

When you remove a view, simply call the dispose() method which will clean everything up for you automatically:

```
var sampleView = new SampleView({model: some_model, collection: some_collection});
sampleView.dispose();
```

(Thanks to JohnnyO for this tip).

How does one handle View disposal on a Parent or Child View? In the last question, we looked at how to effectively dispose views to decreases memory usage (analogous to a type of garbage collection).

Where your application is setup with multiple Parent and Child Views, it is also common to desire removing any DOM elements associated with such views as well as unbinding any event handlers tied to child elements when you no longer require them.

The solution in the last question should be enough to handle this use-case, but if you require a more-explicit example that handles children, we can see one below:

```
Backbone.View.prototype.close = function() {
    if (this.onClose) {
        this.onClose();
    }
    this.remove();
    this.unbind();
};

NewView = Backbone.View.extend({
    initialize: function() {
       this.childViews = [];
    },
    renderChildren: function(item) {
        var itemView = new NewChildView({ model: item });
        $(this.el).prepend(itemView.render());
        this.childViews.push(itemView);
    },
    onClose: function() {
      _(this.childViews).each(function(view) {
        view.close();
      });
    }
});

NewChildView = Backbone.View.extend({
    tagName: 'li',
    render: function() {
    }
});
```

Here, a close() method for views is implemented which disposes of a view when it is no longer needed or needs to be reset. In most cases the view removal should be done at a view layer so that it won't affect any of our models.

For example, if you are working on a blogging application and you remove a view with comments, perhaps another view in your app shows a selection of comments and resetting the collection would affect those views too.

(Thanks to dira for this tip)

What's the best way to combine or append Views to each other? Let us say you have a Collection, where each item in the Collection could itself be a Collection. You can render each item in the Collection, and indeed can render any items which themselves are Collections. The problem you might have is how to render this structure where the HTML reflects the hierarchical nature of the data structure.

The most straight-forward way to approach this problem is to use a framework like Derick Baileys Backbone.Marionette. In this framework is a type of view called a CompositeView.

The basic idea of a CompositeView is that it can render a model and a collection within the same view.

It can render a single model with a template. It can also take a collection from that model and for each model in that collection, render a view. By default it uses the same composite view type that you've defined, to render each of the models in the collection. All you have to do is tell the view instance where the collection is, via the initialize method, and you'll get a recursive hierarchy rendered.

There is a working demo of this in action available online.

And you can get the source code and documentation for Marionette too.

Better Model Property Validation As we learned earlier in the book, the `validate` method on a Model is called before `set` and `save`, and is passed the model attributes updated with the values from these methods.

By default, where we define a custom `validate` method, Backbone passes all of a Model's attributes through this validation each time, regardless of which model attributes are being set.

This means that it can be a challenge to determine which specific fields are being set or validated without being concerned about the others that aren't being set at the same time.

To illustrate this problem better, let us look at a typical registration form use case that:

- Validates form fields using the blur event
- Validates each field regardless of whether other model attributes (aka other form data) are valid or not.

Here is one example of a desired use case:

We have a form where a user focuses and blurs first name, last name, and email HTML input boxes without entering any data. A "this field is required" message should be presented next to each form field.

HTML:

```
<!doctype html>
<html>
<head>
  <meta charset=utf-8>
  <title>Form Validation - Model#validate</title>
  <script src='http://code.jquery.com/jquery.js'></script>
  <script src='http://underscorejs.org/underscore.js'></script>
  <script src='http://backbonejs.org/backbone.js'></script>
</head>
<body>
```

```
  <form>
    <label>First Name</label>
    <input name='firstname'>
    <span data-msg='firstname'></span>
    <br>
    <label>Last Name</label>
    <input name='lastname'>
    <span data-msg='lastname'></span>
    <br>
    <label>Email</label>
    <input name='email'>
    <span data-msg='email'></span>
  </form>
</body>
</html>
```

Some simple validation that could be written using the current Backbone `validate` method to work with this form could be implemented using something like:

```
validate: function(attrs) {

    if(!attrs.firstname) return 'first name is empty';
    if(!attrs.lastname) return 'last name is empty';
    if(!attrs.email) return 'email is empty';

}
```

Unfortunately, this method would trigger a first name error each time any of the fields were blurred and only an error message next to the first name field would be presented.

One potential solution to the problem could be to validate all fields and return all of the errors:

```
validate: function(attrs) {
  var errors = {};

  if (!attrs.firstname) errors.firstname = 'first name is empty';
  if (!attrs.lastname) errors.lastname = 'last name is empty';
  if (!attrs.email) errors.email = 'email is empty';

  if (!_.isEmpty(errors)) return errors;
}
```

This can be adapted into a complete solution that defines a Field model for each input in our form and works within the parameters of our use-case as follows:

```
$(function($) {

  var User = Backbone.Model.extend({
    validate: function(attrs) {
      var errors = this.errors = {};

      if (!attrs.firstname) errors.firstname = 'firstname is required';
      if (!attrs.lastname) errors.lastname = 'lastname is required';
      if (!attrs.email) errors.email = 'email is required';

      if (!_.isEmpty(errors)) return errors;
    }
  });

  var Field = Backbone.View.extend({
    events: {blur: 'validate'},
    initialize: function() {
      this.name = this.$el.attr('name');
      this.$msg = $('[data-msg=' + this.name + ']');
    },
    validate: function() {
      this.model.set(this.name, this.$el.val());
      this.$msg.text(this.model.errors[this.name] || '');
    }
  });

  var user = new User;

  $('input').each(function() {
    new Field({el: this, model: user});
  });

});
```

This works great as the solution checks the validation for each attribute individually and sets the message for the correct blurred field. A demo of the above by @braddunbar is also available.

It unfortunately however forces us to validate all of your form fields every time. If we have multiple client-side validation methods with our particular use case, we may not want to have to call each validation method on every attribute every time, so this solution might not be ideal for everyone.

A potentially better alternative to the above is to use @gfranko's Backbone.validateAll plugin, specifically created to validate specific Model properties (or form fields) without worrying about the validation of any other Model properties (or form fields).

Here is how we would setup a partial User Model and validate method using this plugin, that caters to our use-case:

```
// Create a new User Model
var User = Backbone.Model.extend({

      // RegEx Patterns
      patterns: {

          specialCharacters: '[^a-zA-Z 0-9]+',

          digits: '[0-9]',

          email: '^[a-zA-Z0-9._-]+@[a-zA-Z0-9][a-zA-Z0-9.-]*[.]{1}[a-zA-Z]{2,6}$'
      },

    // Validators
      validators: {

      minLength: function(value, minLength) {
            return value.length >= minLength;

          },

          maxLength: function(value, maxLength) {
            return value.length <= maxLength;

          },

           isEmail: function(value) {
            return User.prototype.validators.pattern(value, User.prototype.patterns.email);

          },

          hasSpecialCharacter: function(value) {
            return User.prototype.validators.pattern(value, User.prototype.patterns.special

          },
         ...
    // We can determine which properties are getting validated by
    // checking to see if properties are equal to null

        validate: function(attrs) {

          var errors = this.errors = {};

          if(attrs.firstname != null) {
              if (!attrs.firstname) {
                  errors.firstname = 'firstname is required';
                  console.log('first name isEmpty validation called');
              }

              else if(!this.validators.minLength(attrs.firstname, 2))
                errors.firstname = 'firstname is too short';
              else if(!this.validators.maxLength(attrs.firstname, 15))
                errors.firstname = 'firstname is too large';
              else if(this.validators.hasSpecialCharacter(attrs.firstname)) errors.firstname
          }

          if(attrs.lastname != null) {

              if (!attrs.lastname) {
                  errors.lastname = 'lastname is required';
                  console.log('last name isEmpty validation called');
              }

              else if(!this.validators.minLength(attrs.lastname, 2))
                errors.lastname = 'lastname is too short';
              else if(!this.validators.maxLength(attrs.lastname, 15))
                errors.lastname = 'lastname is too large';
              else if(this.validators.hasSpecialCharacter(attrs.lastname)) errors.lastname =

          }
```

This allows the logic inside of our validate methods to determine which form fields are currently being set/validated, and does not care about the other model properties that are not trying to be set.

It's fairly straight-forward to use as well. We can simply define a new Model instance and then set the data on our model using the `validateAll` option to use the behavior defined by the plugin:

```
var user = new User();
user.set({ 'firstname': 'Greg' }, {validateAll: false});
```

That's it!.

The Backbone.validateAll logic doesn't override the default Backbone logic by default and so it's perfectly capable of being used for scenarios where you might care more about field-validation performance as well as those where you don't. Both solutions presented in this section should work fine however.

Backbone Extensions

Backbone.Marionette

By Derick Bailey & Addy Osmani

As we've seen, Backbone provides a great set of building blocks for our JavaScript applications. It gives us the core constructs that are needed to build small to mid-sized apps, organize jQuery DOM events, or create single page apps that support mobile devices and large scale enterprise needs. But Backbone is not a complete framework. It's a set of building blocks that leaves much of the application design, architecture and scalability to the developer, including memory management, view management and more.

Backbone.Marionette (or just "Marionette") provides many of the features that the non-trivial application developer needs, above what Backbone itself provides. It is a composite application library that aims to simplify the construction of large scale applications. It does this by providing a collection of common design and implementation patterns found in the applications that the creator, Derick Bailey, and many other contributors have been using to build Backbone apps.

Marionette's key benefits include:

- Scaling applications out with modular, event driven architecture
- Sensible defaults, such as using Underscore templates for view rendering
- Easy to modify to make it work with your application's specific needs
- Reducing boilerplate for views, with specialized view types
- Build on a modular architecture with an Application and modules that attach to it
- Compose your application's visuals at runtime, with Region and Layout
- Nested views and layouts within visual regions
- Built-in memory management and zombie killing in views, regions and layouts
- Built-in event clean up with the EventBinder
- Event-driven architecture with the EventAggregator
- Flexible, "as-needed" architecture allowing you to pick and choose what you need
- And much, much more

Marionette follows a similar philosophy to Backbone in that it provides a suite of components that can be used independently of each other, or used together to create a significant advantages for us as developers. But it steps above the structural components of Backbone and provides an application layer, with more than a dozen components and building blocks.

Marionette's components range greatly in the features they provide, but they all work together to create a composite application layer that can both reduce boilerplate code and provide a much needed application structure. Its core components include:

- **Marionette.Application**: An application object that starts your app via initializers, and more
- **Marionette.Application.module**: Create modules and sub-modules within the application
- **Marionette.AppRouter**: Reduce your routers to nothing more than configuration
- **Marionette.View**: The base View type that other Marionette views extend from (not intended to be used directly)
- **Marionette.ItemView**: A view that renders a single item
- **Marionette.CollectionView**: A view that iterates over a collection, and renders individual `ItemView` instances for each model
- **Marionette.CompositeView**: A collection view and item view, for rendering leaf-branch/composite model hierarchies
- **Marionette.Region**: Manage visual regions of your application, including display and removal of content
- **Marionette.Layout**: A view that renders a layout and creates region managers to manage areas within it
- **Marionette.EventAggregator**: An extension of Backbone.Events, to be used as an event-driven or pub-sub tool
- **Marionette.EventBinder**: An event binding manager, to facilitate binding and unbinding of events
- **Marionette.Renderer**: Render templates with or without data, in a consistent and common manner
- **Marionette.TemplateCache**: Cache templates that are stored in `<script>` blocks, for faster subsequent access
- **Marionette.Callbacks**: Manage a collection of callback methods, and execute them as needed

But like Backbone itself, you're not required to use all of Marionette's components just because you want to use some of them. You can pick and choose which features you want to use, when. This allows you to work with other Backbone frameworks and plugins very easily. It also means that you are not required to engage in an all-or-nothing migration to begin using Marionette.

Boilerplate Rendering Code

Consider the code that it typically requires to render a view with Backbone and Underscore template. We need a template to render, which can be placed in the DOM directly, and we need the JavaScript that defines a view to use the template, and populate that template with data from a model.

```
<script type="text/html" id="my-view-template">
  <div class="row">
    <label>First Name:</label>
    <span><%= firstName %></span>
  </div>
  <div class="row">
    <label>Last Name:</label>
    <span><%= lastName %></span>
  </div>
  <div class="row">
    <label>Email:</label>
    <span><%= email %></span>
  </div>
</script>
</pre>
```

```
var MyView = Backbone.View.extend({
  template: $('#my-view-template').html(),

  render: function(){

    // compile the Underscore.js template
    var compiledTemplate = _.template(this.template);

    // render the template with the model data
    var data = this.model.toJSON();
    var html = compiledTemplate(data);

    // populate the view with the rendered html
    this.$el.html(html);
  }
});
```

Once this is in place, you need to create an instance of your view and pass your model in to it. Then you can take the view's `el` and append it to the DOM in order to display the view.

```
var myModel = new MyModel({
  firstName: 'Derick',
  lastName: 'Bailey',
  email: 'derickbailey@gmail.com'
});

var myView = new MyView({
  model: myModel
})

myView.render();

$('#content').html(myView.el)
```

This is a standard set up for defining, building, rendering, and displaying a view with Backbone. This is also what we call "boilerplate code" - code that is repeated over and over and over again, across every project and every implementation of the same functionality. It gets to be very tedious and repetitious very quickly.

Enter Marionette's `ItemView` - a simple way to reduce the boilerplate of defining a view.

Reducing Boilerplate With Marionette.ItemView

All of Marionette's view types - with the exception of `Marionette.View` - include a built-in `render` method that handles the core rendering logic for you. By changing the `MyView` instance from `Backbone.View` then, we can take advantage of this. Instead of having to provide our own `render` method for the view, we can let Marionette render it for us. We'll still use the same Underscore.js template and rendering mechanism, but the implementation of this is hidden behind the scenes for us. Thus, we can reduce the amount of code needed for this view.

```
var MyView = Marionette.ItemView.extend({
  template: '#my-view-template'
});
```

And that's it - that's all you need to get the exact same behaviour as the previous view implementation. Just replace `Backbone.View.extend` with `Marionette.ItemView.extend`, then get rid of the `render` method. You can still create the view instance with a `model`, call the `render` method on the view instance, and display the view in the DOM the same way that we did before. But the view definition has been reduced to a single line of configuration for the template.

Memory Management

In addition to the reduction of code needed to define a view, Marionette includes some advanced memory management in all of it's views, making the job of cleaning up a view instance and it's event handlers, easy.

Consider the following view implementation:

```
var ZombieView = Backbone.View.extend({
  template: '#my-view-template',

  initialize: function(){

    // bind the model change to re-render this view
    this.model.on('change', this.render, this);

  },

  render: function(){

    // This alert is going to demonstrate a problem
    alert('We`re rendering the view');

  }
});
```

If we create two instances of this view using the same variable name for both instances, and then change a value in the model, how many times will we see the alert box?

```
var myModel = new MyModel({
  firstName: 'Jeremy',
  lastName: 'Ashkenas',
  email: 'jeremy@example.com'
});

// create the first view instance
var zombieView = new ZombieView({
  model: myModel
})

// create a second view instance, re-using
// the same variable name to store it
zombieView = new ZombieView({
  model: myModel
})

myModel.set('email', 'jeremy@gmail.com');
```

Since we're re-using the save `zombieView` variable for both instances, the first instance of the view will fall out of scope immediately after the second is created. This allows the JavaScript garbage collector to come along and clean it up, which should mean the first view instance is no longer active and no longer going to respond to the model's "change" event.

But when we run this code, we end up with the alert box showing up twice!

The problem is caused by the model event binding in the view's `initialize` method. Whenever we pass `this.render` as the callback method to the model's `on` event binding, the model itself is being given a direct reference to the view instance. Since the model is now holding a reference to the view instance, replacing the `zombieView` variable with a new view instance is not going to let the original view fall out of scope. The model still has a reference, therefore the view is still in scope.

Since the original view is still in scope, and the second view instance is also in scope, changing data on the model will cause both view instances to respond.

Fixing this is easy, though. You just need to call `off` when the view is done with it's work and ready to be closed. To do this, add a `close` method to the view.

```
var ZombieView = Backbone.View.extend({
  template: '#my-view-template',

  initialize: function(){
    // bind the model change to re-render this view
    this.model.on('change', this.render, this);
  },

  close: function(){
    this.model.off('change', this.render, this);
  },

  render: function(){

    // This alert is going to demonstrate a problem
    alert('We`re rendering the view');

  }
});
```

Then call `close` on the first instance when it is no longer needed, and only one view instance will remain alive.

```
var myModel = new MyModel({
  firstName: 'Jeremy',
  lastName: 'Ashkenas',
  email: 'jeremy@example.com'
});

// create the first view instance
var zombieView = new ZombieView({
  model: myModel
})
zombieView.close(); // double-tap the zombie

// create a second view instance, re-using
// the same variable name to store it
zombieView = new ZombieView({
  model: myModel
})

myModel.set('email', 'jeremy@gmail.com');
```

Now we only see once alert box when this code runs.

Rather than having to manually remove these event handlers, though, we can let Marionette do it for us.

```
var ZombieView = Marionette.ItemView.extend({
  template: '#my-view-template',

  initialize: function(){

    // bind the model change to re-render this view
    this.bindTo(this.model, 'change', this.render, this);

  },

  render: function(){

    // This alert is going to demonstrate a problem
    alert('We`re rendering the view');

  }
});
```

Notice in this case we are using a method called `bindTo`. This method comes from Marionette's `EventBinder` object, and is added on to all of Marionette's view types. The `bindTo` method signature is similar to that of the `on` method, with the exception of passing the object that triggers the event as the first parameter.

Marionette's views also provide a `close` event, in which the event bindings that are set up with the `bindTo` are automatically removed. This means we no longer need to define a `close` method directly, and when we use the `bindTo` method, we know that our events will be removed and our views will not turn in to zombies.

But how do we automate the call to `close` on a view, in the real application? When and where do we call that? Enter the `Marionette.Region` - an object that manages the lifecycle of an individual view.

Region Management

After a view is created, it typically needs to be placed in the DOM so that it becomes visible. This is usually done with a jQuery selector and setting the `html()` of the resulting object:

```
var myModel = new MyModel({
  firstName: 'Jeremy',
  lastName: 'Ashkenas',
  email: 'jeremy@gmail.com'
});

var myView = new MyView({
  model: myModel
})

myView.render();

// show the view in the DOM
$('#content').html(myView.el)
```

This, again, is boilerplate code. We shouldn't have to manually call `render` and manually select the DOM elements to show the view. Furthermore, this code doesn't lend itself to closing any previous view instance that might be attached to the DOM element we want to populate. And we've seen the danger of zombie views already.

To solve these problems, Marionette provides a `Region` object - an object that manages the lifecycle of individual views, displayed in a particular DOM element.

```
// create a region instance, telling it which DOM element to manage
var myRegion = new Marionette.Region({
  el: '#content'
});
```

```
// show a view in the region
var view1 = new MyView({ /* ... */ });
myRegion.show(view1);

// somewhere else in the code,
// show a different view
var view2 = new MyView({ /* ... */ });
myRegion.show(view2);
```

There are several things to note, here. First, we're telling the region what DOM element to manage by specifying an `el` in the region instance. Second, we're no longer calling the `render` method on our views. And lastly, we're not calling `close` on our view, either, though this is getting called for us.

When we use a region to manage the lifecycle of our views, and display the views in the DOM, the region itself handles these concerns. By passing a view instance in to the `show` method of the region, it will call the render method on the view for us. It will then take the resulting `el` of the view and populate the DOM element.

The next time we call the `show` method of the region, the region remembers that it is currently displaying a view. The region calls the `close` method on the view, removes it from the DOM, and then proceeds to run the render & display code for the new view that was passed in.

Since the region handles calling `close` for us, and we're using the `bindTo` event binder in our view instance, we no longer have to worry about zombie views in our application.

Marionette Todo app

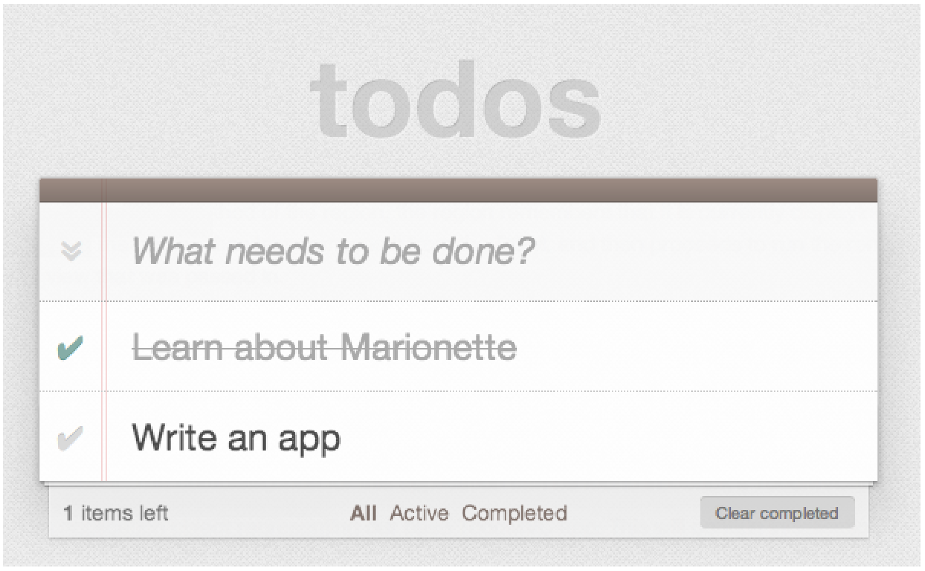
Having learned about Marionette's high-level concepts, let's explore refactoring the Todo application we created in our first practical to use it. The complete code for this application can be found in Derick's TodoMVC fork.

Our final implementation will be visually and functionally equivalent to the original app, as seen below.

First, we define an application object representing our base TodoMVC app. This will contain initialisation code and define the default layout regions for our app.

TodoMVC.js:

```
var TodoMVC = new Marionette.Application();

TodoMVC.addRegions({
  header : '#header',
```



```
  main   : '#main',
  footer : '#footer'
});

TodoMVC.on('initialize:after', function(){
  Backbone.history.start();
});
```

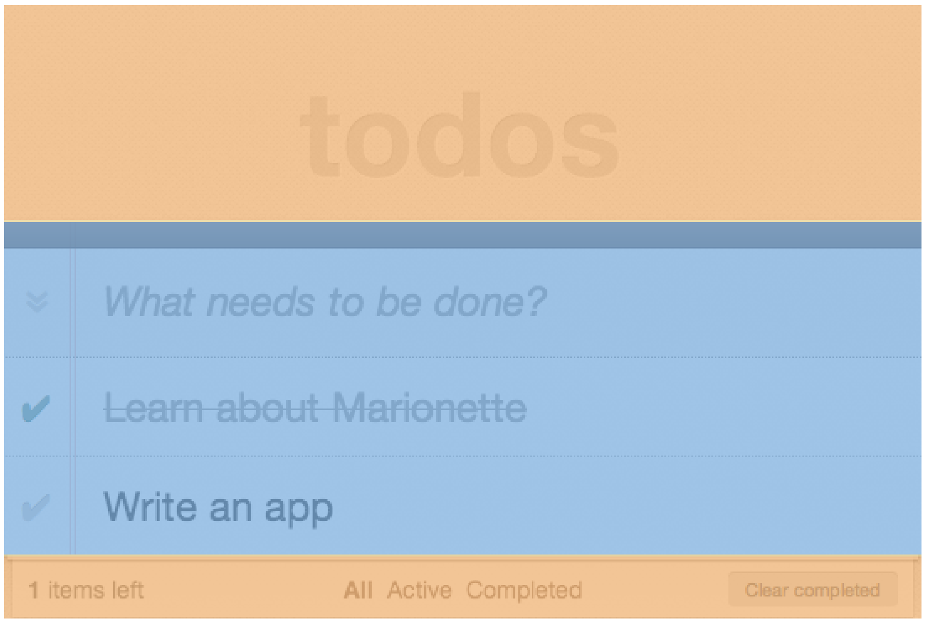
Regions are used to manage the content that's displayed within specific elements, and the `addRegions` method on the `TodoMVC` object is just a shortcut for creating `Region` objects. We supply a jQuery selector for each region to manage (e.g `#header`, `#main` and `#footer`) and then tell the region to show various Backbone views within that region.

Once the application object has been initialised, we call `Backbone.history.start()` to route the initial URL.

Next, we define our Layouts. A layout is a specialised type of view that extends from `Marionette.ItemView` directly. This means its intended to render a single template and may or may not have a model (or `item`) associated with the template.

One of the main differences between a Layout and an `ItemView` is that the layout contains regions. When defining a Layout, we supply it with a `template` but also the regions that the template contains. After rendering the layout, we can display other views within the layout using the regions that were defined.

In our TodoMVC Layout module below, we define Layouts for:



- Header: where we can create new Todos
- Footer: where we summarise how many Todos are remaining/have been completed

This captures some of the view logic that was previously in our `AppView` and `TodoView`.

Note that Marionette modules (such as the below) offer a simple module system which are used to create privacy and encapsulation in Marionette apps. These certainly don't have to be used however, and later on in this section we'll provide links to alternative implementations using RequireJS + AMD instead.

TodoMVC.Layout.js:

```
TodoMVC.module('Layout', function(Layout, App, Backbone, Marionette, $, _){

  // Layout Header View
  // ------------------

  Layout.Header = Marionette.ItemView.extend({
    template : '#template-header',

    // UI bindings create cached attributes that
    // point to jQuery selected objects
    ui : {
```

```
    input : '#new-todo'
  },

  events : {
    'keypress #new-todo':   'onInputKeypress'
  },

  onInputKeypress : function(evt) {
    var ENTER_KEY = 13;
    var todoText = this.ui.input.val().trim();

    if ( evt.which === ENTER_KEY && todoText ) {
      this.collection.create({
        title : todoText
      });
      this.ui.input.val('');
    }
  }
});

// Layout Footer View
// ------------------

Layout.Footer = Marionette.Layout.extend({
  template : '#template-footer',

  // UI bindings create cached attributes that
  // point to jQuery selected objects
  ui : {
    count   : '#todo-count strong',
    filters : '#filters a'
  },

  events : {
    'click #clear-completed' : 'onClearClick'
  },

  initialize : function() {
    this.bindTo(App.vent, 'todoList:filter', this.updateFilterSelection, this);
    this.bindTo(this.collection, 'all', this.updateCount, this);
  },

  onRender : function() {
    this.updateCount();
  },
    updateCount : function() {
      var count = this.collection.getActive().length;
      this.ui.count.html(count);

      if (count === 0) {
        this.$el.parent().hide();
      } else {
        this.$el.parent().show();
      }
    },

    updateFilterSelection : function(filter) {
      this.ui.filters
        .removeClass('selected')
        .filter('[href="#' + filter + '"]')
        .addClass('selected');
    },

    onClearClick : function() {
      var completed = this.collection.getCompleted();
      completed.forEach(function destroy(todo) {
        todo.destroy();
      });
    }
  });

});
```

Next, we tackle application routing and workflow, such as controlling Layouts in the page which can be shown or hidden.

Marionette uses the concept of an AppRouter to simplify routing. This reduces the boilerplate for handling route events and allows routers to be configured to call methods on an object directly. We configure our AppRouter using `appRoutes`.

This replaces the `'*filter': 'setFilter'` route defined in our original Workspace router, seen below:

```
        var Workspace = Backbone.Router.extend({
                routes:{
                        '*filter': 'setFilter'
                },

                setFilter: function( param ) {
                        // Set the current filter to be used
```

```
                    window.app.TodoFilter = param.trim() || '';

                    // Trigger a collection reset/addAll
                    window.app.Todos.trigger('reset');
                }
        });
```

The TodoList Controller, also found in this next code block, handles some of the remaining visibility logic originally found in `AppView` and `TodoView`, albeit using very readable Layouts.

TodoMVC.TodoList.js:

```
TodoMVC.module('TodoList', function(TodoList, App, Backbone, Marionette, $, _){

  // TodoList Router
  // ---------------
  //
  // Handle routes to show the active vs complete todo items

  TodoList.Router = Marionette.AppRouter.extend({
    appRoutes : {
      '*filter': 'filterItems'
    }
  });

  // TodoList Controller (Mediator)
  // ------------------------------
  //
  // Control the workflow and logic that exists at the application
  // level, above the implementation detail of views and models

  TodoList.Controller = function(){
    this.todoList = new App.Todos.TodoList();
  };

  _.extend(TodoList.Controller.prototype, {

    // Start the app by showing the appropriate views
    // and fetching the list of todo items, if there are any
    start: function(){
      this.showHeader(this.todoList);
      this.showFooter(this.todoList);
      this.showTodoList(this.todoList);

      this.todoList.fetch();
    },

    showHeader: function(todoList){
      var header = new App.Layout.Header({
        collection: todoList
      });
      App.header.show(header);
    },

    showFooter: function(todoList){
      var footer = new App.Layout.Footer({
        collection: todoList
      });
      App.footer.show(footer);
    },

    showTodoList: function(todoList){
      App.main.show(new TodoList.Views.ListView({
        collection : todoList
      }));
    },

    // Set the filter to show complete or all items
    filterItems: function(filter){
      App.vent.trigger('todoList:filter', filter.trim() || '');
    }
  });

  // TodoList Initializer
  // --------------------
  //
  // Get the TodoList up and running by initializing the mediator
  // when the the application is started, pulling in all of the
  // existing Todo items and displaying them.

  TodoList.addInitializer(function(){

    var controller = new TodoList.Controller();
    new TodoList.Router({
      controller: controller
    });

    controller.start();

  });
```

```
});
```

Controllers In this particular app, note that Controllers don't add a great deal to the overall workflow. In general however, Marionette's philosophy on routers is that they should be an after-thought in the implementation of applications. Quite often, we'll see many bad examples of developers abusing Backbone's routing system by making it the sole controller of the entire application workflow and logic.

This inevitably leads to mashing every possible combination of code in to the router methods - view creation, model loading, coordinating different parts of the app, etc. Developers such as Derick views this as a violation of the single-responsibility principle (SRP) and separation of concerns.

Backbone's router and history exists to deal with a specific aspect of browsers - managing the forward and back buttons. Marionette feels it should be limited to that, with the code that gets executed by the navigation being somewhere else. This allows the application to be used with or without a router. We can call a controller's "show" method from a button click, from an application event handler, or from a router, and we will end up with the same application state no matter how we called that method.

Derick has written extensively about his thoughts on this topic, which you can read more about on his blog:

- http://lostechies.com/derickbailey/2011/12/27/the-responsibilities-of-the-various-pieces-of-backbone-js/
- http://lostechies.com/derickbailey/2012/01/02/reducing-backbone-routers-to-nothing-more-than-configur
- http://lostechies.com/derickbailey/2012/02/06/3-stages-of-a-backbone-applications-startup/

CompositeView We then get to defining the actual views for individual Todo items and lists of items in our TodoMVC application. For this, we make use of Marionette's `CompositeViews`. The idea behind a CompositeView is that it represents a visualisation of a composite or hierarchical structure of leaves (or nodes) and branches.

Think of these views as being a hierarchy of parent-child models, and recursive by default. For each item in a collection that the composite view is handling the same CompositeView type will be used to render the item. For non-recursive hierarchies, though, we are able to override the item view by defining an `itemView` attribute.

For our Todo List Item View, we define it as an ItemView, then our Todo List View is a CompositeView where we override the `itemView` setting and tell it to use the Todo List item View for each item in the collection.

TodoMVC.TodoList.Views.js

```
TodoMVC.module('TodoList.Views', function(Views, App, Backbone, Marionette, $, _){

  // Todo List Item View
  // -------------------
  //
  // Display an individual todo item, and respond to changes
  // that are made to the item, including marking completed.

  Views.ItemView = Marionette.ItemView.extend({
    tagName : 'li',
      template : '#template-todoItemView',

      ui : {
        edit : '.edit'
      },

      events : {
        'click .destroy' : 'destroy',
        'dblclick label' : 'onEditClick',
        'keypress .edit' : 'onEditKeypress',
        'click .toggle'  : 'toggle'
      },

      initialize : function() {
        this.bindTo(this.model, 'change', this.render, this);
      },

      onRender : function() {
        this.$el.removeClass('active completed');
        if (this.model.get('completed')) this.$el.addClass('completed');
        else this.$el.addClass('active');
      },

      destroy : function() {
        this.model.destroy();
      },

      toggle  : function() {
        this.model.toggle().save();
      },

      onEditClick : function() {
        this.$el.addClass('editing');
        this.ui.edit.focus();
      },
      onEditKeypress : function(evt) {
        var ENTER_KEY = 13;
        var todoText = this.ui.edit.val().trim();

        if ( evt.which === ENTER_KEY && todoText ) {
          this.model.set('title', todoText).save();
          this.$el.removeClass('editing');
        }
      }
  });

  // Item List View
  // --------------
  //
  // Controls the rendering of the list of items, including the
  // filtering of active vs completed items for display.

  Views.ListView = Marionette.CompositeView.extend({
    template : '#template-todoListCompositeView',
      itemView : Views.ItemView,
      itemViewContainer : '#todo-list',

      ui : {
        toggle : '#toggle-all'
      },

      events : {
        'click #toggle-all' : 'onToggleAllClick'
      },

      initialize : function() {
        this.bindTo(this.collection, 'all', this.update, this);
      },

      onRender : function() {
        this.update();
      },

      update : function() {
        function reduceCompleted(left, right) { return left && right.get('completed'); }
        var allCompleted = this.collection.reduce(reduceCompleted,true);
        this.ui.toggle.prop('checked', allCompleted);

        if (this.collection.length === 0) {
          this.$el.parent().hide();
        } else {
          this.$el.parent().show();
        }
      },

      onToggleAllClick : function(evt) {
        var isChecked = evt.currentTarget.checked;
        this.collection.each(function(todo){
          todo.save({'completed': isChecked});
        });
      }
  });

  // Application Event Handlers
  // --------------------------
  //
  // Handler for filtering the list of items by showing and
  // hiding through the use of various CSS classes

  App.vent.on('todoList:filter',function(filter) {
    filter = filter || 'all';
    $('#todoapp').attr('class', 'filter-' + filter);
  });

});
```

At the end of the last code block, you will also notice an event handler using `vent`. This is an event aggregator that allows us to handle `filterItem` triggers from our TodoList controller.

Finally, we define the model and collection for representing our Todo items. These are semantically not very different from the original versions we used in our first practical and have been re-written to better fit in with Derick's preferred style of coding.

Todos.js:

```
TodoMVC.module('Todos', function(Todos, App, Backbone, Marionette, $, _){

  // Todo Model
  // ----------

  Todos.Todo = Backbone.Model.extend({
    localStorage: new Backbone.LocalStorage('todos-backbone'),

    defaults: {
      title     : '',
      completed : false,
      created   : 0
    },

    initialize : function() {
      if (this.isNew()) this.set('created', Date.now());
    },

    toggle  : function() {
      return this.set('completed', !this.isCompleted());
    },

    isCompleted: function() {
      return this.get('completed');
    }
  });

  // Todo Collection
  // ---------------

  Todos.TodoList = Backbone.Collection.extend({
    model: Todos.Todo,

    localStorage: new Backbone.LocalStorage('todos-backbone'),

    getCompleted: function() {
      return this.filter(this._isCompleted);
    },

    getActive: function() {
      return this.reject(this._isCompleted);
    },

    comparator: function( todo ) {
      return todo.get('created');
    },

    _isCompleted: function(todo){
      return todo.isCompleted();
    }
  });

});
```

We finally kick-start everything off in our application index file, by calling `start` on our main application object:

Initialisation:

```
$(function(){
  // Start the TodoMVC app (defined in js/TodoMVC.js)
  TodoMVC.start();
});
```

And that's it!

Is the Marionette implementation of the Todo app more maintainable?

Derick feels that maintainability largely comes down to modularity, separating responsibilities (SRP and SoC) and other related patterns for keeping concerns from being mixed together. It can however be difficult to simply extract things in to separate modules for the sake of extraction, abstraction, or dividing the concept down in to it's most finite parts.

The Single Responsibility Principle (SRP) tells us quite the opposite - that we need to understand the context in which things change. What parts always change together, in *this* system? What parts can change independently? Without knowing this, we won't know what pieces should be broken out in to separate components and modules, vs put together in to the same module or object.

The way Derick organizes his apps into modules is by creating a breakdown of concepts at each level. A higher level module is a higher level of concern - an aggregation of responsibilities. Each responsibility is broken down in to an expressive API set that is implemented by lower level modules (Dependency Inversion Principle). These are coordinated through a mediator - which he typically refers to as the Controller in a module.

The way that Derick organizes his files also plays directly into maintainability and he has also written up posts about the importance of keeping a sane application folder structure that I recommend reading:

- http://lostechies.com/derickbailey/2012/02/02/javascript-file-folder-structures-just-pick-one/
- http://hilojs.codeplex.com/discussions/362875#post869640

Marionette And Flexibility

Marionette is a flexible framework, much like Backbone itself. It offers a wide variety of tools to help create and organize an application architecture on top of Backbone, but like Backbone itself, it doesn't dictate that you have to use all of it's pieces in order to use any of them.

The flexibility and versatility in Marionette is easiest to understand by examining three variations of TodoMVC that have been created for comparison purposes:

- Simple - by Jarrod Overson
- RequireJS - also by Jarrod
- Marionette modules - by Derick Bailey

The simple version: This version of TodoMVC shows some raw use of Marionette's various view types, an application object, and the event aggregator. The objects that are created are added directly to the global namespace and are fairly straightforward. This is a great example of how Marionette can be used to augment existing code without having to re-write everything around Marionette.

The RequireJS version: Using Marionette with RequireJS helps to create a modularized application architecture - a tremendously important concept in scaling JavaScript applications. RequireJS provides a powerful set of tools that can be leveraged to great advantage, making Marionette even more flexible than it already is.

The Marionette module version: RequireJS isn't the only way to create a modularized application architecture, though. For those that wish to build applications in modules and namespaces, Marionette provides a built-in module and namespacing structure. This example application takes the simple version of the application and re-writes it in to a namespaced application architecture, with an application controller (mediator / workflow object) that brings all of the pieces together.

Marionette certainly provides its share of opinions in how a Backbone application should be architected. The combination of modules, view types, event aggregator, application objects, and more, can be used to create a very powerful and flexible architecture based on these opinions.

But as you can see, Marionette isn't a completely rigid, "my way or the highway" framework. It provides many elements of an application foundation that can be mixed and matched with other architectural styles, such as AMD or namespacing, or provide simple augmentation to existing projects by reducing boilerplate code for rendering views.

This flexibility creates a much greater opportunity for Marionette to provide value to you and your projects, as it allows you to scale the use of Marionette with your application's needs.

And So Much More

This is just the tip of the proverbial ice-berg for Marionette, even for the `ItemView` and `Region` objects that we've explored. There is far more functionality, more features, and more flexibility and customizability that can be put to use in both of these objects. Then we have the other dozen or so components that Marionette provides, each with their own set of behaviors built in, customization and extension points, and more.

To learn more about Marionette, it's components, the features they provide and how to use them, check out the Marionette documentation, links to the wiki, to the source code, the project core contributors, and much more at http://marionettejs.com.

Thorax

By Ryan Eastridge & Addy Osmani

Part of Backbone's appeal is that it provides structure but is generally unopinionated, in particular when it comes to views. Thorax makes an opinionated decision to use Handlebars as it's templating solution. Some of the patterns found in Marionette are found in Thorax as well. Marionette exposes most of these patterns as JavaScript APIs while in Thorax they are often exposed as template helpers. This chapter assumes the reader has knowledge of Handlebars.

Thorax was created by Ryan Eastridge and Kevin Decker to create Walmart's mobile web application. This chapter is limited to Thorax's templating features and patterns implemented in Thorax that you can utilize in your application regardless of wether you choose to adopt Thorax. To learn more about other features implemented in Thorax and to download boilerplate projects visit the Thorax website.

Hello World

`Thorax.View` differs from `Backbone.View` in that there is no `options` object. All arguments passed to the constructor become properties of the view, which in turn become available to the `template`:

```
var view = new Thorax.View({
    greeting: 'Hello',
    template: '{{greeting}} World!'
});
view.appendTo('body');
```

In most examples in this chapter a `template` property will be specified. In larger projects including the boilerplate projects provided on the Thorax website a `name` property would instead be used and a `template` of the same file name in your project would automatically be assigned to the view.

If a `model` is set on a view, it's attributes also become available to the template:

```
var view = new Thorax.View({
    model: new Thorax.Model({key: 'value'}),
    template: '{{key}}'
});
```

Embedding child views

The view helper allows you to embed other views within a view. Child views can be specified as properties of the view:

```
var parent = new Thorax.View({
    child: new Thorax.View(...),
    template: '{{view child}}'
});
```

Or the name of a child view to initialize (and any optional properties to pass). In this case the child view must have previously been created with `extend` and a `name` property:

```
var ChildView = Thorax.View.extend({
    name: 'child',
    template: ...
});

var parent = new Thorax.View({
    template: '{{view "child" key="value"}}'
});
```

The view helper may also be used as a block helper, in which case the block will be assigned as the `template` property of the child view:

```
{{#view child}}
    child will have this block
    set as it's template property
{{/view}}
```

Handlebars is a string based, while `Backbone.View` instances have a DOM `el`. Since we are mixing metaphors, the embedding of views works via a placeholder mechanism where the `view` helper in this case adds the view passed to the helper to a hash of `children`, then injects placeholder HTML into the template such as:

```
<div data-view-placeholder-cid="view2"></div>
```

Then once the parent view is rendered, we walk the DOM in search of all the placeholders we created, replacing them with the child views' `el`s:

```
this.$el.find('[data-view-placeholder-cid]').forEach(function(el) {
    var cid = el.getAttribute('data-view-placeholder-cid'),
        view = this.children[cid];
    view.render();
    $(el).replaceWith(view.el);
}, this);
```

View helpers

One of the most useful constructs in Thorax is `Handlebars.registerViewHelper` (which differs from `Handlebars.registerHelper`). This method will register a new block helper that will create and embed a `HelperView` instance with it's `template` set to the captured block. A `HelperView` instance is different from that of a regular child view in that it's context will be that of the parent's in the template. Like other child views it will have a `parent` property set to that of the declaring view. Many of the built in helpers in Thorax including the `collection` helper are created in this manner.

A simple example would be an `on` helper that re-rendered the generated `HelperView` instance each time an event was triggered on the declaring / parent view:

```
Handlebars.registerViewHelper('on', function(eventName, helperView) {
    helperView.parent.on(eventName, function() {
        helperView.render();
    });
});
```

An example use of this would be to have a counter that would incriment each time a button was clicked. This example makes use of the `button` helper in Thorax which simply makes a button that calls a method when clicked:

```
{{#on "incrimented"}}{{i}}{/on}}
{{#button trigger="incrimented"}}Add{{/button}}
```

And the corresponding view class:

```
new Thorax.View({
    events: {
        incrimented: function() {
            ++this.i;
        }
    },
    initialize: function() {
        this.i = 0;
    },
    template: ...
});
```

collection helper

The `collection` helper creates and embeds a `CollectionView` instance, creating a view for each item in a collection, updating when items are added, removed or changed in the collection. The simplest usage of the helper would look like:

```
{{#collection kittens}}
  <li>{{name}}</li>
{{/collection}}
```

And the corresponding view:

```
new Thorax.View({
  kittens: new Thorax.Collection(...),
  template: ...
});
```

The block in this case will be assigned as the `template` for each item view created, and the context will be the `attributes` of the given model. This helper accepts options that can be arbitrary HTML attributes, a `tag` option to specify the type of tag containing the collection, or any of the following:

- `item-template` - A template to display for each model. If a block is specified it will become the item-template
- `item-view` - A view class to use when each item view is created
- `empty-template` - A template to display when the collection is empty. If an inverse / else block is specified it will become the empty-template
- `empty-view` - A view to display when the collection is empty

Options and blocks can be used in combination, in this case creating a `KittenView` class with a `template` set to the captured block for each kitten in the collection:

```
{{#collection kittens item-view="KittenView" tag="ul"}}
  <li>{{name}}</li>
```

```
{{else}}
  <li>No kittens!</li>
{{/collection}}
```

Note that multiple collections can be used per view, and collections can be nested. This is useful when there are models that contain collections that contain models that contain…

```
{{#collection kittens}}
  <h2>{{name}}</h2>
  <p>Kills:</p>
  {{#collection miceKilled tag="ul"}}
    <li>{{name}}</li>
  {{/collection}}
{{/collection}}
```

Custom HTML data attributes

Thorax makes heavy use of custom data attributes to operate. While some make sense only within the context of Thorax, several are quite useful to have in any Backbone project for writing other functions against, or for general debugging. In order to add some to your views in non Thorax projects, override the `setElement` method in your base view class:

```
MyApplication.View = Backbone.View.extend({
  setElement: function() {
      var response = Backbone.View.prototype.setElement.apply(this, arguments);
      this.name && this.$el.attr('data-view-name', this.name);
      this.$el.attr('data-view-cid', this.cid);
      this.collection && this.$el.attr('data-collection-cid', this.collection.cid);
      this.model && this.$el.attr('data-model-cid', this.model.cid);
      return response;
  }
});
```

In addition making your application more immediately comprehensible in the inspector, it's now possible to extend jQuery / Zepto with functions to lookup the closest view, model or collection to a given element. In order to make it work save references to each view created in your base view class by overriding the `_configure` method:

```
MyApplication.View = Backbone.View.extend({
    _configure: function() {
        Backbone.View.prototype._configure.apply(this, arguments);
```

```
            Thorax._viewsIndexedByCid[this.cid] = cid;
        },
        dispose: function() {
            Backbone.View.prototype.dispose.apply(this, arguments);
            delete Thorax._viewsIndexedByCid[this.cid];
        }
    });
```

Then we can extend jQuery / Zepto:

```
    $.fn.view = function() {
        var el = $(this).closest('[data-view-cid]');
        return el && Thorax._viewsIndexedByCid[el.attr('data-view-cid')];
    };

    $.fn.model = function(view) {
        var $this = $(this),
            modelElement = $this.closest('[data-model-cid]'),
            modelCid = modelElement && modelElement.attr('[data-model-cid]');
        if (modelCid) {
            var view = $this.view();
            return view && view.model;
        }
        return false;
    };
```

Now instead of storing references to models randomly throughout your application to lookup when a given DOM event occurs you can use `$(element).model()`. In Thorax, this can particularly useful in conjunction with the `collection` helper which generates a view class (with a `model` property) for each `model` in the collection. An example template:

```
    {{#collection kittens tag="ul"}}
      <li>{{name}}</li>
    {{/collection}}
```

And the corresponding view class:

```
    Thorax.View.extend({
      events: {
        'click li': function(event) {
          var kitten = $(event.target).model();
          console.log('Clicked on ' + kitten.get('name'));
        }
      },
      kittens: new Thorax.Collection(...),
      template: ...
    });
```

A common anti-pattern in Backbone applications is to assign a `className` to a single view class. Consider using the `data-view-name` attribute as a CSS selector instead, saving CSS classes for things that will be used multiple times:

```
[data-view-name="child"] {

}
```

Thorax Resources

No Backbone related tutorial would be complete without a todo application. A Thorax implementation of TodoMVC is available, in addition to this far simpler example composed of this single handlebars template:

```
{{#collection todos tag="ul"}}
  <li{{#if done}} class="done"{{/if}}>
    <input type="checkbox" name="done"{{#if done}} checked="checked"{{/if}}>
    <span>{{item}}</span>
  </li>
{{/collection}}
<form>
  <input type="text">
  <input type="submit" value="Add">
</form>
```

and the corresponding JavaScript:

```
var todosView = Thorax.View({
    todos: new Thorax.Collection(),
    events: {
        'change input[type="checkbox"]': function(event) {
            var target = $(event.target);
            target.model().set({done: !!target.attr('checked')});
        },
        'submit form': function(event) {
            event.preventDefault();
            var input = this.$('input[type="text"]');
            this.todos.add({item: input.val()});
            input.val('');
        }
    },
    template: '...'
});
todosView.appendTo('body');
```

To see Thorax in action on a large scale website visit walmart.com on any Android or iOS device. For a complete list of resources visit the Thorax website.

Modular Development

Introduction

When we say an application is modular, we generally mean it's composed of a set of highly decoupled, distinct pieces of functionality stored in modules. As you probably know, loose coupling facilitates easier maintainability of apps by removing dependencies where possible. When this is implemented efficiently, its quite easy to see how changes to one part of a system may affect another.

Unlike some more traditional programming languages however, the current iteration of JavaScript (ECMA-262) doesn't provide developers with the means to import such modules of code in a clean, organized manner. It's one of the concerns with specifications that haven't required great thought until more recent years where the need for more organized JavaScript applications became apparent.

Instead, developers at present are left to fall back on variations of the module or object literal patterns. With many of these, module scripts are strung together in the DOM with namespaces being described by a single global object where it's still possible to incur naming collisions in your architecture. There's also no clean way to handle dependency management without some manual effort or third party tools.

Whilst native solutions to these problems will be arriving in ES Harmony, the good news is that writing modular JavaScript has never been easier and you can start doing it today.

In this next part of the book, we're going to look at how to use AMD modules and RequireJS for cleanly wrapping units of code in your application into manageable modules, and an alternate approach using routes to determine when modules are loaded.

Organizing modules with RequireJS and AMD

Partly Contributed by Jack Franklin

RequireJS is a popular script loader written by James Burke - a developer who has been quite instrumental in helping shape the AMD module format, which we'll discuss more shortly. Some of RequireJS's capabilities include helping to load multiple script files, helping define modules with or without dependencies and loading in non-script dependencies such as text files.

Maintainability problems with multiple script files

You might be thinking that there is little benefit to RequireJS. After all, you can simply load in your JavaScript files through multiple `<script>` tags, which is very straightforward. However, doing it that way has a lot of drawbacks, namely the HTTP overhead.

Everytime the browser loads in a file you've referenced in a `<script>` tag, it makes a HTTP request to load the file's contents. It has to make a new HTTP request for each file you want to load, which causes problems.

- Browsers are limited in how many parallel requests they can make, so often it's slow to load multiple files, as it can only do a certain number at a time. This number depends on the user's settings and browser, but is usually around 4-8. When working on Backbone applications it's good to split your app into multiple JS files, so it's easy to hit that limit quickly. This can be negated by minifying your code into one file as part of a build process, but does not help with the next point.
- When a script is loaded it is done so synchronously. This means that while it's being loaded, the browser cannot continue rendering the page.

What tools like RequireJS do is load in scripts asynchronously. This means we have to adjust our code slightly, you can't just swap out `<script>` elements for a small piece of RequireJS code, but the benefits are very worthwhile.

- Loading in the scripts asychrnously means they are non-blocking. The browser can continue to render the rest of the page as the scripts are being loaded, speeding up the initial load time.
- We can load modules in more intelligently, having more control over when they are loaded; along with making sure modules that have dependencies are loaded in the right order so all dependencies are met.

Need for better dependency management

Dependency management is a challenging subject, in particular when writing JavaScript in the browser. The closest thing we have to dependency management by default is simply making sure we order our `<script>` tags such that code that depends on code in another file is loaded after. This is not a good approach. As I've already discussed, loading multiple files in that way is bad for performance; needing them to be loaded in a certain order is very brittle.

Being able to load different code in only when needed is something RequireJS is very good at. Rather than load all our JavaScript code in at run-time, often a better approach is to dynamically load modules at run-time, only when that code is required. This avoids loading all the code in when the user first hits your application, consequently speeding up the initial load times.

Think about the GMail web-client for a moment. When users initially load up the page on their first visit, Google can simply hide widgets such as the chat module until a user has indicated (by clicking 'expand') that they wish to use it. Through dynamic dependency loading, Google could load up the chat module only then, rather than forcing all users to load it when the page first initializes. This can improve performance and load times and can definitely prove useful when building larger applications. As the codebase for an application grows this becomes even more important.

The important thing to note here is that while it's absolutely fine to develop applications without a script loader, there are significant benefits to utilising tools like RequireJS in your application.

Asynchrous Module Definition (AMD)

RequireJS implements the AMD Specification which defines a method for writing modular code and managing dependencies. The RequireJS website also has a section documenting the reasons behind implementing AMD:

> The AMD format comes from wanting a module format that was better than today's "write a bunch of script tags with implicit dependencies that you have to manually order" and something that was easy to use directly in the browser. Something with good debugging characteristics that did not require server-specific tooling to get started.

Writing AMD modules with RequireJS

As discussed above, the overall goal for the AMD format is to provide a solution for modular JavaScript that developers can use today. The two key concepts you need to be aware of when using it with a script-loader are a `define()` method for facilitating module definition and a `require()` method for handling dependency loading. `define()` is used to define named or unnamed modules based on the proposal using the following signature:

```
define(
    module_id /*optional*/,
    [dependencies] /*optional*/,
    definition function /*function for instantiating the module or object*/
);
```

As you can tell by the inline comments, the `module_id` is an optional argument which is typically only required when non-AMD concatenation tools are being used (there may be some other edge cases where it's useful too). When this argument is left out, we call the module 'anonymous'. When working with anonymous modules, the idea of a module's identity is DRY, making it trivial to avoid duplication of filenames and code.

Back to the define signature, the dependencies argument represents an array of dependencies which are required by the module you are defining and the third argument ('definition function') is a function that's executed to instantiate your module. A barebone module (compatible with RequireJS) could be defined using `define()` as follows:

```
// A module ID has been omitted here to make the module anonymous

define(['foo', 'bar'],
    // module definition function
    // dependencies (foo and bar) are mapped to function parameters
    function ( foo, bar ) {
        // return a value that defines the module export
        // (i.e the functionality we want to expose for consumption)

        // create your module here
        var myModule = {
            doStuff:function(){
                console.log('Yay! Stuff');
            }
        }

        return myModule;
});
```

Alternate syntax There is also a sugared version of `define()` available that allows you to declare your dependencies as local variables using `require()`. This will feel familiar to anyone who's used node, and can be easier to add or remove dependencies. Here is the previous snippet using the alternate syntax:

```
// A module ID has been omitted here to make the module anonymous
```

```
define(function(require){
        // module definition function
    // dependencies (foo and bar) are defined as local vars
    var foo = require('foo'),
        bar = require('bar');

    // return a value that defines the module export
    // (i.e the functionality we want to expose for consumption)

    // create your module here
    var myModule = {
        doStuff:function(){
            console.log('Yay! Stuff');
        }
    }

    return myModule;
});
```

The `require()` method is typically used to load code in a top-level JavaScript file or within a module should you wish to dynamically fetch dependencies. An example of its usage is:

```
// Consider 'foo' and 'bar' are two external modules
// In this example, the 'exports' from the two modules loaded are passed as
// function arguments to the callback (foo and bar)
// so that they can similarly be accessed

require(['foo', 'bar'], function ( foo, bar ) {
        // rest of your code here
        foo.doSomething();
});
```

Addy's post on Writing Modular JS covers the AMD specification in much more detail. Defining and using modules will be covered in this book shortly when we look at more structured examples of using RequireJS.

Getting Started with RequireJS

Before using RequireJS and Backbone we will first set up a very basic RequireJS project to demonstrate how it works. The first thing to do is to Download RequireJS. When you load in the RequireJS script in your HTML file, you need to also tell it where your main JavaScript file exists. Typically this will be called something like "app.js", and is the main entry point for your application. You do this by passing in a `data-main` attribute:

```
<script data-main="app.js" src="lib/require.js"></script>
```

Now, RequireJS will automatically load in `app.js` for you.

RequireJS Configuration In your main JavaScript file that you pass in through the `data-main` attribute, you can configure RequireJS. This is done by calling `require.config`, and passing in an object:

```
require.config({
    // your configuration key/values here
    baseUrl: "app", // generally the same directory as the script used in a data-main attri
    paths: {}, // set up custom paths to libraries, or paths to RequireJS plugins
    shim: {}, // used for setting up all Shims (see below for more detail)
});
```

The main reason you'd want to configure RequireJS is for shims, which is used to allow RequireJS to work with libraries that don't use `define()` in the code, but expose themselves globally. We'll cover this shortly. To see other configuration options available to you, I recommend checking out the RequireJS documentation.

RequireJS Shims To use a library with RequireJS, ideally that library should come with AMD support. That is, it uses the `define` method to define the library as a module. However, some libraries - including Backbone and one of its dependencies, Underscore - don't do this. Fortunately RequireJS comes with a way to bypass this.

To demonstrate this, first let's shim Underscore, and then we'll shim Backbone too. Shims are very simple to implement:

```
require.config({
    shim: {
        'lib/underscore': {
          exports: '_'
        }
    }
});
```

The important line there is `exports:  '_'`. That tells RequireJS that whenever we require the file `'lib/underscore'`, it would return the global _ object, as that's where Underscore exports itself too, it adds the _ property to the global object. Now when we require Underscore, RequireJS will return that _ property. Now we can set up a shim for Backbone too:

```
require.config({
    shim: {
        'lib/underscore': {
          exports: '_'
        },
        'lib/backbone': {
            deps: ['lib/underscore', 'jquery'],
            exports: 'Backbone'
        }
    }
});
```

Again, that configuration tells RequireJS to return the global `Backbone` property that Backbone exports, but also this time you'll notice that Backbone's dependencies are defined. This means whenever this:

```
require( 'lib/backbone', function( Backbone ) {...} );
```

Is run, it will first make sure the dependencies are met, and then pass the global `Backbone` object into the callback function. You don't need to do this with every library, only the ones that don't support AMD. For example, jQuery does support it, as of jQuery 1.7.

If you'd like to read more about general RequireJS usage, the RequireJS API docs are incredibly thorough and easy to read.

Custom Paths Typing long paths to file names like `lib/backbone` can get tedious. RequireJS lets us set up custom paths in our configuration object. Here, whenever I refer to "underscore", RequireJS will look for the file `lib/underscore.js`:

```
require.config({
    paths: {
        'underscore': 'lib/underscore'
    }
});
```

Of course, this can be combined with a shim:

```
require.config({
    paths: {
        'underscore': 'lib/underscore'
    },
    shim: {
        'underscore': {
          exports: '_'
        }
    }
});
```

Just make sure that in your shim settings, you refer to the custom path too. Now you can do:

```
require( ['underscore'], function(_) {
// code here
});
```

To shim Underscore but still use a custom path.

Require.js and Backbone Examples

Now that we've taken a look at how to define AMD modules, let's review how to go about wrapping components like views and collections so that they can also be easily loaded as dependencies for any parts of your application that require them. At it's simplest, a Backbone model may just require Backbone and Underscore.js. These are dependencies, so we can define those when defining the new modules. Note that the following examples presume you have configured RequireJS to shim Backbone and Underscore, as discussed previously.

Wrapping modules, views and other components with AMD For example, here is how a model is defined.

```
define(['underscore', 'backbone'], function(_, Backbone) {
  var myModel = Backbone.Model.extend({

    // Default attributes
    defaults: {
      content: 'hello world',
    },

    // A dummy initialization method
    initialize: function() {
    },

    clear: function() {
      this.destroy();
      this.view.remove();
    }

  });
  return myModel;
});
```

Note how we alias Underscore.js's instance to `_` and Backbone to just `Backbone`, making it very trivial to convert non-AMD code over to using this module format. For a view which might require other dependencies such as jQuery, this can similarly be done as follows:

```
define([
  'jquery',
  'underscore',
  'backbone',
  'collections/mycollection',
  'views/myview'
  ], function($, _, Backbone, myCollection, myView){

  var AppView = Backbone.View.extend({
  ...
```

Aliasing to the dollar-sign (`$`), once again makes it very easy to encapsulate any part of an application you wish using AMD.

Doing it this way makes it easy to organise your Backbone application as you like. It's recommended to separate modules into folders. For example, individual folders for models, collections, views and so on. RequireJS doesn't care about what folder structure you use; as long as you use the correct path when using `require`, it will happily pull in the file.

If you'd like to take a look at how others do it, Pete Hawkins' Backbone Stack repository is a good example of structuring a Backbone application, using RequireJS. Greg Franko has also written an overview of how he uses Backbone and Require, and Jeremy Kahn's post neatly describes his approach. For a full look at a sample application, the Backbone and Require example of the TodoMVC repository should be your starting point.

Keeping Your Templates External Using RequireJS And The Text Plugin

Moving your [Underscore/Mustache/Handlebars] templates to external files is actually quite straight-forward. As this application makes use of RequireJS, I'll discuss how to implement external templates using this specific script loader.

RequireJS has a special plugin called text.js which is used to load in text file dependencies. To use the text plugin, simply follow these simple steps:

1. Download the plugin from http://requirejs.org/docs/download.html#text and place it in either the same directory as your application's main JS file or a suitable sub-directory.

2. Next, include the text.js plugin in your initial RequireJS configuration options. In the code snippet below, we assume that RequireJS is being included in our page prior to this code snippet being executed.

```
require.config( {
    paths: {
        'text': 'libs/require/text',
    },
    baseUrl: 'app'
} );
```

3. When the `text!` prefix is used for a dependency, RequireJS will automatically load the text plugin and treat the dependency as a text resource. A typical example of this in action may look like:

```
require(['js/app', 'text!templates/mainView.html'],
    function( app, mainView ) {
        // the contents of the mainView file will be
        // loaded into mainView for usage.
    }
);
```

4. Finally we can use the text resource that's been loaded for templating purposes. You're probably used to storing your HTML templates inline using a script with a specific identifier.

With Underscore.js's micro-templating (and jQuery) this would typically be:

HTML:

```
<script type="text/template" id="mainViewTemplate">
    <% _.each( person, function( person_item ){ %>
        <li><%= person_item.get('name') %></li>
    <% }); %>
</script>
```

JS:

```
var compiled_template = _.template( $('#mainViewTemplate').html() );
```

With RequireJS and the text plugin however, it's as simple as saving your template into an external text file (say, `mainView.html`) and doing the following:

```
require(['js/app', 'text!templates/mainView.html'],
    function(app, mainView){
        var compiled_template = _.template( mainView );
    }
);
```

That's it! Now you can apply your template to a view in Backbone with something like:

```
collection.someview.$el.html( compiled_template( { results: collection.models } ) );
```

All templating solutions will have their own custom methods for handling template compilation, but if you understand the above, substituting Underscore's micro-templating for any other solution should be fairly trivial.

Note: You may also be interested in looking at RequireJS tpl. It's an AMD-compatible version of the Underscore templating system that also includes support for optimization (pre-compiled templates) which can lead to better performance and no evals. I have yet to use it myself, but it comes as a recommended resource.

Optimizing Backbone apps for production with the RequireJS Optimizer

As experienced developers may know, an essential final step when writing both small and large JavaScript web applications is the build process. The majority of non-trivial apps are likely to consist of more than one or two scripts and so optimizing, minimizing and concatenating your scripts prior to pushing them to production will require your users to download a reduced number (if not just one) script file.

If this is your first time looking at build scripts, Addy Osmani's screencast on build scripts may be useful.

Another benefit to using RequireJS is its command line optimization tool, R.js. This has a number of capabilities, including:

- Concatenating specific scripts and minifying them using external tools such as UglifyJS (which is used by default) or Google's Closure Compiler for optimal browser delivery, whilst preserving the ability to dynamically load modules
- Optimizing CSS and stylesheets by inlining CSS files imported using @import, stripping out comments etc.

- The ability to run AMD projects in both Node and Rhino (more on this later)

You'll notice that I mentioned the word 'specific' in the first bullet point. The RequireJS optimizer only concatenates module scripts that have been specified in arrays of string literals passed to top-level (i.e non-local) require and define calls. As clarified by the optimizer docs this means that Backbone modules defined like this:

```
define(['jquery','backbone','underscore', 'collections/sample','views/test'],
    function($,Backbone, _, Sample, Test){
        //...
    });
```

will combine fine, however inline dependencies such as:

```
var models = someCondition ? ['models/ab','models/ac'] : ['models/ba','models/bc'];
```

will be ignored. This is by design as it ensures that dynamic dependency/module loading can still take place even after optimization.

Although the RequireJS optimizer works fine in both Node and Java environments, it's strongly recommended to run it under Node as it executes significantly faster there. In my experience, it's a piece of cake to get setup with either environment, so go for whichever you feel most comfortable with.

To get started with r.js, grab it from the RequireJS download page or through NPM. Now, the RequireJS optimizer works absolutely fine for single script and CSS files, but for most cases you'll want to actually optimize an entire Backbone project. You *could* do this completely from the command-line, but a cleaner option is using build profiles.

Below is an example of a build file taken from the modular jQuery Mobile app referenced later in this book. A **build profile** (commonly named `app.build.js`) informs RequireJS to copy all of the content of `appDir` to a directory defined by `dir` (in this case `../release`). This will apply all of the necessary optimizations inside the release folder. The `baseUrl` is used to resolve the paths for your modules. It should ideally be relative to `appDir`.

Near the bottom of this sample file, you'll see an array called `modules`. This is where you specify the module names you wish to have optimized. In this case we're optimizing the main application called 'app', which maps to `appDir/app.js`. If we had set the `baseUrl` to 'scripts', it would be mapped to `appDir/scripts/app.js`.

```
({
    appDir: './',
    baseUrl: './',
    dir: '../release',
    paths: {
       'backbone':          'libs/AMDbackbone-0.5.3',
        'underscore':       'libs/underscore-1.2.2',
        'jquery':           'libs/jQuery-1.7.1',
        'json2':            'libs/json2',
        'datepicker':       'libs/jQuery.ui.datepicker',
        'datepickermobile': 'libs/jquery.ui.datepicker.mobile',
        'jquerymobile':     'libs/jquery.mobile-1.0'
    },
    optimize: 'uglify',
    modules: [
        {
            name: 'app',
            exclude: [
                // If you prefer not to include certain libs exclude them here
            ]
        }
    ]
})
```

The way the build system in r.js works is that it traverses app.js (whatever modules you've passed) and resolved dependencies, concatenating them into the final `release`(dir) folder. CSS is treated the same way.

The build profile is usually placed inside the 'scripts' or 'js' directory of your project. As per the docs, this file can however exist anywhere you wish, but you'll need to edit the contents of your build profile accordingly.

Finally, to run the build, execute the following command once inside your `appDir` or `appDir/scripts` directory:

```
node ../../r.js -o app.build.js
```

That's it. As long as you have UglifyJS/Closure tools setup correctly, r.js should be able to easily optimize your entire Backbone project in just a few key-strokes. If you would like to learn more about build profiles, James Burke has a heavily commented sample file with all the possible options available.

Optimize and Build a Backbone.js JavaScript application with RequireJS using Packages

Contributed by Bill Heaton

When a JavaScript application is too complex or large to build in a single file, grouping the application's components into packages allows for script dependencies to download in parallel, and facilitates only loading **packaged** and other modular code as the site experience requires the specific set of dependencies.

RequireJS, the (JavaScript) module loading library, has an optimizer to build a JavaScript-based application and provides various options. A build profile is the recipe for your build, much like a build.xml file is used to build a project with ANT. The benefit of building with **r.js** not only results in speedy script loading with minified code, but also provides a way to package components of your application.

- Optimizing one JavaScript file
- Optimizing a whole project
- Optimizing a project in layers or packages

In a complex application, organizing code into *packages* is an attractive build strategy. The build profile in this article is based on an test application currently under development (files list below). The application framework is built with open source libraries. The main objective in this build profile is to optimize an application developed with Backbone.js using modular code, following the Asynchronous Module Definition (AMD) format. AMD and RequireJS provide the structure for writing modular code with dependencies. Backbone.js provides the code organization for developing models, views and collections and also interactions with a RESTful API.

Below is an outline of the applications file organization, followed by the build profile to build modular (or packaged) layers a JavaScript driven application.

File organization Assume the following directories and file organization, with app.build.js as the build profile (a sibling to both source and release directories). Note that the files in the list below named *section* can be any component of the application, e.g. *header*, *login*)

```
.-- app.build.js
|-- app-release
`-- app-src
    |-- collections
    |   |-- base.js
    |   |-- sections-segments.js
    |   `-- sections.js
    |-- docs
    |   `--docco.css
    |-- models
    |   |-- base.js
    |   |-- branding.js
    |   `-- section.js
    |-- packages
    |   |-- header
    |   |   |-- models
    |   |   |   |-- nav.js
    |   |   |   `-- link.js
    |   |   |-- templates
    |   |   |   |-- branding.js
    |   |   |   |-- nav.js
    |   |   |   `-- links.js
    |   |   `-- views
    |   |       |-- nav.js
    |   |       |-- branding.js
    |   |       `-- link.js
    |   |-- header.js
    |   `-- ... more packages here e.g. cart, checkout ...
    |-- syncs
    |   |-- rest
    |   |   `-- sections.js
    |   |-- factory.js
    |   `-- localstorage.js
    |-- test
    |   |-- fixtures
    |   |   `-- sections.json
    |   |-- header
    |   |   |-- index.html
    |   |   `-- spec.js
    |   |-- lib
    |   |   `-- Jasmine
    |   |-- models
    |   |-- utils
    |   |-- global-spec.js
    |-- utils
    |   |-- ajax.js
    |   |-- baselib.js
    |   |-- debug.js
    |   |-- localstorage.js
    |   `-- shims.js
    |-- vendor
    |-- |-- backbone-min.js
    |   |-- jquery.min.js
    |   |-- jquery.mobile-1.0.min.js
    |   |-- json2.js
    |   |-- modernizr.min.js
    |   |-- mustache.js
    |   |-- require.js
    |   |-- text.js
    |   `-- underscore.js
    |-- views
    |   |-- base.js
    |   `-- collection.js
    |-- application.js
    |-- collections.js
    |-- index.html
    |-- main.js
    |-- models.js
    |-- syncs.js
    |-- utils.js
    |-- vendor.js
    `-- views.js
```

Build profile to optimize modular dependencies with code organized in packages The build profile can be organized to divide parallel downloads for various sections of the application.

This strategy demonstrated builds common or site-wide groups of (core) *models*, *views*, collections which are extended from a base.js constructor which extends the appropriate backbone method, e.g. Backbone.Model. The *packages* directory organizes code by section / responsibility, e.g. cart, checkout, etc. Notice that within the example *header* package the directory structure is similar to the app root directory file structure. A *package* (of modularized code) has dependencies from the common libraries in your application and also has specific code for the packages execution alone; other packages should not require another packages dependencies. A *utils* directory has shims, helpers, and common library code to support the application. A *syncs* directory to define persistence with your RESTful api and/or localStorage. The *vendor* libraries folder will not be built, there is no need to do so, you may decide to use a CDN (then set these paths to : *empty:*). And finally a *test* directory for Jasmine unit test specs, which may be ignored in the build as well if you choose.

Also notice the there are .js files named the same as the directories, these are the files listed in the paths. these are strategic to group sets of files to build, examples follow the build profile below.

```
({
    appDir: './app-src',
    baseUrl: './',
    dir: './app-build',
    optimize: 'uglify',
    paths: {
        // will not build 3rd party code, it's already built
        'text'         : 'vendor/text',
        'json2'        : 'vendor/json2.min',
        'modernizr'    : 'vendor/modernizr.min',
        'jquery'       : 'vendor/jquery-1.7.1',
        'jquerymobile' : 'vendor/jquery.mobile.min.js',
        'underscore'   : 'vendor/underscore',
        'mustache'     : 'vendor/mustache',
        'backbone'     : 'vendor/backbone',
        // files that define dependencies...
        // ignore vendor libraries, but need a group to do so
        'vendor'       : 'vendor',
        // application modules/packages these files define dependencies
        // and may also group modules into objects if needed to require
        // by groups rather than individual files
        'utils'        : 'utils',
        'models'       : 'models',
        'views'        : 'views',
        'collections'  : 'collections',
        // packages to build
        'header'       : 'packages/header'
        //... more packages
    },
    modules: [
        // Common libraries, Utilities, Syncs, Models, Views, Collections
        {
            name: 'utils',
            exclude: ['vendor']
        },
        {
            name: 'syncs',
            exclude: ['vendor', 'utils']
        },
        {
            name: 'models',
            exclude: ['vendor', 'utils', 'syncs']
        },
        {
            name: 'views',
            exclude: ['vendor', 'utils', 'syncs', 'models']
        },
        {
            name: 'collections',
            exclude: ['vendor', 'utils', 'syncs', 'models', 'views']
        },
        // Packages
        {
            name: 'header',
            exclude: ['vendor', 'utils', 'syncs', 'models', 'views', 'collections']
        }
        // ... and so much more ...
    ]
})
```

The above build profile is designed for balancing scalability and performance.

Examples of the grouped sets of code dependencies

The contents of the vendor.js which is not built into a package may use some *no conflict* calls as well.

```
// List of vendor libraries, e.g. jQuery, Underscore, Backbone, etc.
// this module is used with the r.js optimizer tool during build
// @see <http://requirejs.org/docs/faq-optimization.html>
define([ 'jquery', 'underscore', 'backbone', 'modernizr', 'mustache' ],
function ($,        _,           Backbone,   Modernizr,   Mustache) {
    // call no conflicts so if needed you can use multiple versions of $
    $.noConflict();
    _.noConflict();
    Backbone.noConflict();
});
```

For your application common library code.

```
// List of utility libraries,
define([ 'utils/ajax', 'utils/baselib', 'utils/localstorage', 'utils/debug', 'utils/shims' ]
function (ajax,         baselib,         localstorage,         debug) {
    return {
        'ajax' : ajax,
        'baselib' : baselib,
        'localstorage' : localstorage,
        'debug' : debug
    };
    // the shim only extend JavaScript when needed, e.g. Object.create
});
```

An example where you intend to use require the common models in another package file.

```
// List of models
// models in this directory are intended for site-wide usage
// grouping site-wide models in this module (object)
// optimizes the performance and keeps dependencies organized
// when the (build) optimizer is run.
define([ 'models/branding', 'models/section' ],
function (Branding,          Section) {
    return {
        'Branding' : Branding,
        'Section'  : Section
    };
});
```

A quick note on code standards Notice that in the above examples the parameters may begin with lower or upper case characters. The variable names uses in the parameters that begin with *Uppercase* are *Constructors* and the *lowercase* variable names are not, they may be instances created by a constructor, or perhaps an object or function that is not meant to used with *new*.

The convention recommended is to use Upper CamelCase for constructors and lower camelCase for others.

Common Pitfall when organizing code in modules Be careful not define circular dependencies. For example, in a common *models* package (models.js) dependencies are listed for the files in your models directory

```
define([ 'models/branding', 'models/section' ], function (branding, section)
// ...
return { 'branding' : branding, 'section', section }
```

Then when another packages requires a common model you can access the models objects returned from your common models.js file like so…

```
define([ 'models', 'utils' ], function (models, utils) {
var branding = models.branding, debug = utils.debug;
```

Perhaps after using the model a few times you get into the habit of requiring "model". Later you need add another common model with extends a model you already defined. So the pitfall begins, you add a new model inside your models directory and add a reference this same model in the model.js:

```
define([ 'models/branding', 'models/section', 'models/section-b' ], function (branding, sect
// ...
return { 'branding' : branding, 'section', section, 'section-b' : section-b }
```

However in your *models/section-b.js* file you define a dependency using the model.js which returns the models in an object like so…

```
define([ 'models' ], function (models, utils) {
var section = models.section;
```

Above is the mistake in models.js a dependency was added for models/section-b and in section-b a dependency is defined for model. The new models/section-b.js requires *model* and model.js requires *models/section-b.js* - a circular dependency. This should result in a load timeout error from RequireJS, but not tell you about the circular dependency.

For other common mistakes see the COMMON ERRORS page on the RequireJS site.

Executing the Build with r.js If you intalled r.js with Node's npm (package manager) like so…

```
> npm install requirejs
```

…you can execute the build on the command line:

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

Exercise: Building a modular Backbone app with AMD & RequireJS

In this chapter, we'll look at our first practical Backbone & RequireJS project - how to build a modular Todo application. The application will allow us to add new todos, edit new todos and clear todo items that have been marked as completed. For a more advanced practical, see the section on mobile Backbone development.

The complete code for the application can can be found in the `practicals/modular-todo-app` folder of this repo (thanks to Thomas Davis and Jérôme Gravel-Niquet). Alternatively grab a copy of my side-project TodoMVC which contains the sources to both AMD and non-AMD versions.

Note: Thomas may be covering a practical on this exercise in more detail on backbonetutorials.com at some point soon, but for this section I'll be covering what I consider the core concepts.

Overview

Writing a 'modular' Backbone application can be a straight-forward process. There are however, some key conceptual differences to be aware of if opting to use AMD as your module format of choice:

- As AMD isn't a standard native to JavaScript or the browser, it's necessary to use a script loader (such as RequireJS or curl.js) in order to support defining components and modules using this module format. As we've already reviewed, there are a number of advantages to using the AMD as well as RequireJS to assist here.
- Models, views, controllers and routers need to be encapsulated *using* the AMD-format. This allows each component of our Backbone application to cleanly manage dependencies (e.g collections required by a view) in the same way that AMD allows non-Backbone modules to.
- Non-Backbone components/modules (such as utilities or application helpers) can also be encapsulated using AMD. I encourage you to try developing these modules in such a way that they can both be used and tested independent of your Backbone code as this will increase their ability to be re-used elsewhere.

Now that we've reviewed the basics, let's take a look at developing our application. For reference, the structure of our app is as follows:

```
index.html
...js/
    main.js
    .../models
            todo.js
    .../views
            app.js
            todos.js
    .../collections
            todos.js
    .../templates
            stats.html
            todos.html
    ../libs
        .../backbone
        .../jquery
        .../underscore
        .../require
                require.js
                text.js
...css/
```

Markup

The markup for the application is relatively simple and consists of three primary parts: an input section for entering new todo items (`create-todo`), a list section to display existing items (which can also be edited in-place) (`todo-list`) and finally a section summarizing how many items are left to be completed (`todo-stats`).

```
<div id="todoapp">

      <div class="content">

        <div id="create-todo">
          <input id="new-todo" placeholder="What needs to be done?" type="text" />
          <span class="ui-tooltip-top">Press Enter to save this task</span>
        </div>

        <div id="todos">
          <ul id="todo-list"></ul>
        </div>

        <div id="todo-stats"></div>

      </div>

</div>
```

The rest of the tutorial will now focus on the JavaScript side of the practical.

Configuration options

If you've read the earlier chapter on AMD, you may have noticed that explicitly needing to define each dependency a Backbone module (view, collection or other module) may require with it can get a little tedious. This can however be improved.

In order to simplify referencing common paths the modules in our application may use, we use a RequireJS configuration object, which is typically defined as a top-level script file. Configuration objects have a number of useful capabilities, the most useful being mode name-mapping. Name-maps are basically a key:value pair, where the key defines the alias you wish to use for a path and the value represents the true location of the path.

In the code-sample below, you can see some typical examples of common name-maps which include: `backbone`, `underscore`, `jquery` and depending on your choice, the RequireJS `text` plugin, which assists with loading text assets like templates.

main.js

```
require.config({
  baseUrl:'../',
  paths: {
    jquery: 'libs/jquery/jquery-min',
    underscore: 'libs/underscore/underscore-min',
    backbone: 'libs/backbone/backbone-optamd3-min',
    text: 'libs/require/text'
  }
});

require(['views/app'], function(AppView){
  var app_view = new AppView;
});
```

The `require()` at the end of our main.js file is simply there so we can load and instantiate the primary view for our application (`views/app.js`). You'll commonly see both this and the configuration object included in most top-level script files for a project.

In addition to offering name-mapping, the configuration object can be used to define additional properties such as `waitSeconds` - the number of seconds to wait before script loading times out and `locale`, should you wish to load up i18n bundles for custom languages. The `baseUrl` is simply the path to use for module lookups.

For more information on configuration objects, please feel free to check out the excellent guide to them in the RequireJS docs.

Modularizing our models, views and collections

Before we dive into AMD-wrapped versions of our Backbone components, let's review a sample of a non-AMD view. The following view listens for changes to its model (a Todo item) and re-renders if a user edits the value of the item.

```
var TodoView = Backbone.View.extend({

    //... is a list tag.
    tagName:  'li',

    // Cache the template function for a single item.
    template: _.template($('#item-template').html()),

    // The DOM events specific to an item.
    events: {
      'click .check'              : 'toggleDone',
      'dblclick div.todo-content' : 'edit',
      'click span.todo-destroy'   : 'clear',
      'keypress .todo-input'      : 'updateOnEnter'
    },

    // The TodoView listens for changes to its model, re-rendering. Since there's
    // a one-to-one correspondence between a **Todo** and a **TodoView** in this
    // app, we set a direct reference on the model for convenience.
    initialize: function() {
      this.model.on('change', this.render, this);
      this.model.view = this;
    },
    ...
```

Note how for templating the common practice of referencing a script by an ID (or other selector) and obtaining its value is used. This of course requires that the template being accessed is implicitly defined in our markup. The following is the 'embedded' version of our template being referenced above:

```
<script type="text/template" id="item-template">
      <div class="todo <%= done ? 'done' : '' %>">
        <div class="display">
          <input class="check" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
          <div class="todo-content"></div>
          <span class="todo-destroy"></span>
        </div>
        <div class="edit">
          <input class="todo-input" type="text" value="" />
        </div>
      </div>
</script>
```

Whilst there is nothing wrong with the template itself, once we begin to develop larger applications requiring multiple templates, including them all in our markup on page-load can quickly become both unmanageable and come with performance costs. We'll look at solving this problem in a minute.

Let's now take a look at the AMD-version of our view. As discussed earlier, the 'module' is wrapped using AMD's `define()` which allows us to specify the dependencies our view requires. Using the mapped paths to 'jquery' etc. simplifies referencing common dependencies and instances of dependencies are themselves mapped to local variables that we can access (e.g 'jquery' is mapped to `$`).

views/todo.js

```
define([
  'jquery',
  'underscore',
  'backbone',
  'text!templates/todos.html'
  ], function($, _, Backbone, todosTemplate){
  var TodoView = Backbone.View.extend({

    //... is a list tag.
    tagName:  'li',

    // Cache the template function for a single item.
    template: _.template(todosTemplate),

    // The DOM events specific to an item.
    events: {
      'click .check'              : 'toggleDone',
      'dblclick div.todo-content' : 'edit',
      'click span.todo-destroy'   : 'clear',
      'keypress .todo-input'      : 'updateOnEnter'
    },

    // The TodoView listens for changes to its model, re-rendering. Since there's
    // a one-to-one correspondence between a **Todo** and a **TodoView** in this
    // app, we set a direct reference on the model for convenience.
    initialize: function() {
      this.model.on('change', this.render, this);
      this.model.view = this;
    },

    // Re-render the contents of the todo item.
    render: function() {
      this.$el.html(this.template(this.model.toJSON()));
      this.setContent();
      return this;
    },

    // Use `jQuery.text` to set the contents of the todo item.
    setContent: function() {
      var content = this.model.get('content');
      this.$('.todo-content').text(content);
      this.input = this.$('.todo-input');
      this.input.on('blur', this.close);
      this.input.val(content);
    },
    ...
```

From a maintenance perspective, there's nothing logically different in this version of our view, except for how we approach templating.

Using the RequireJS text plugin (the dependency marked `text`), we can actually store all of the contents for the template we looked at earlier in an external file (todos.html).

templates/todos.html

```
<div class="todo <%= done ? 'done' : '' %>">
    <div class="display">
      <input class="check" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
      <div class="todo-content"></div>
      <span class="todo-destroy"></span>
    </div>
    <div class="edit">
      <input class="todo-input" type="text" value="" />
    </div>
</div>
```

There's no longer a need to be concerned with IDs for the template as we can map its contents to a local variable (in this case `todosTemplate`). We then simply pass this to the Underscore.js templating function `_.template()` the same way we normally would have the value of our template script.

Next, let's look at how to define models as dependencies which can be pulled into collections. Here's an AMD-compatible model module, which has two default values: a `content` attribute for the content of a Todo item and a boolean `done` state, allowing us to trigger whether the item has been completed or not.

models/todo.js

```
define(['underscore', 'backbone'], function(_, Backbone) {
  var TodoModel = Backbone.Model.extend({

    // Default attributes for the todo.
    defaults: {
      // Ensure that each todo created has `content`.
      content: 'empty todo...',
      done: false
    },

    initialize: function() {
    },

    // Toggle the `done` state of this todo item.
    toggle: function() {
      this.save({done: !this.get('done')});
    },

    // Remove this Todo from *localStorage* and delete its view.
    clear: function() {
      this.destroy();
      this.view.remove();
    }

  });
  return TodoModel;
});
```

As per other types of dependencies, we can easily map our model module to a local variable (in this case `Todo`) so it can be referenced as the model to use for our `TodosCollection`. This collection also supports a simple `done()` filter for narrowing down Todo items that have been completed and a `remaining()` filter for those that are still outstanding.

collections/todos.js

```
define([
  'underscore',
  'backbone',
  'libs/backbone/localstorage',
  'models/todo'
  ], function(_, Backbone, Store, Todo){

    var TodosCollection = Backbone.Collection.extend({

    // Reference to this collection's model.
    model: Todo,

    // Save all of the todo items under the `todos` namespace.
    localStorage: new Store('todos'),

    // Filter down the list of all todo items that are finished.
    done: function() {
      return this.filter(function(todo){ return todo.get('done'); });
    },

    // Filter down the list to only todo items that are still not finished.
    remaining: function() {
      return this.without.apply(this, this.done());
    },
    ...
```

In addition to allowing users to add new Todo items from views (which we then insert as models in a collection), we ideally also want to be able to display how many items have been completed and how many are remaining. We've already defined filters that can provide us this information in the above collection, so let's use them in our main application view.

views/app.js

```
define([
  'jquery',
  'underscore',
  'backbone',
  'collections/todos',
  'views/todo',
  'text!templates/stats.html'
  ], function($, _, Backbone, Todos, TodoView, statsTemplate){

  var AppView = Backbone.View.extend({

    // Instead of generating a new element, bind to the existing skeleton of
    // the App already present in the HTML.
    el: $('#todoapp'),

    // Our template for the line of statistics at the bottom of the app.
    statsTemplate: _.template(statsTemplate),

    // ...events, initialize() etc. can be seen in the complete file

    // Re-rendering the App just means refreshing the statistics -- the rest
    // of the app doesn't change.
    render: function() {
      var done = Todos.done().length;
      this.$('#todo-stats').html(this.statsTemplate({
        total:      Todos.length,
        done:       Todos.done().length,
        remaining:  Todos.remaining().length
      }));
    },
    ...
```

Above, we map the second template for this project, `templates/stats.html` to `statsTemplate` which is used for rendering the overall `done` and `remaining` states. This works by simply passing our template the length of our overall Todos collection (`Todos.length` - the number of Todo items created so far) and similarly the length (counts) for items that have been completed (`Todos.done().length`) or are remaining (`Todos.remaining().length`).

The contents of our `statsTemplate` can be seen below. It's nothing too complicated, but does use ternary conditions to evaluate whether we should state there's "1 item" or "2 items" in a particular state.

```
<% if (total) { %>
        <span class="todo-count">
          <span class="number"><%= remaining %></span>
          <span class="word"><%= remaining == 1 ? 'item' : 'items' %></span> left.
        </span>
      <% } %>
      <% if (done) { %>
        <span class="todo-clear">
          <a href="#">
            Clear <span class="number-done"><%= done %></span>
            completed <span class="word-done"><%= done == 1 ? 'item' : 'items' %></span>
          </a>
        </span>
      <% } %>
```

The rest of the source for the Todo app mainly consists of code for handling user and application events, but that rounds up most of the core concepts for this practical.

To see how everything ties together, feel free to grab the source by cloning this repo or browse it online to learn more. I hope you find it helpful!.

Note: While this first practical doesn't use a build profile as outlined in the chapter on using the RequireJS optimizer, we will be using one in the section on building mobile Backbone applications.

Route based module loading

This section will discuss a route based approach to module loading as implemented in Lumbar by Kevin Decker. Like RequireJS, Lumbar is also a modular build system, but the pattern it implements for loading routes may be used with any build system.

The specifics of the Lumbar build tool are not discussed in this book. To see a complete Lumbar based project with the loader and build system see Thorax which provides boilerplate projects for various environments including Lumbar.

JSON based module configuration

RequireJS defines dependencies per file, while Lumbar defines a list of files for each module in a central JSON configuration file, outputting a single JavaScript file for each defined module. Lumbar requires that each module (except the base module) define a single router and a list of routes. An example file might look like:

```
 {
    "modules": {
        "base": {
            "scripts": [
                "js/lib/underscore.js",
                "js/lib/backbone.js",
                "etc"
            ]
        },
        "pages": {
            "scripts": [
                "js/routers/pages.js",
                "js/views/pages/index.js",
                "etc"
            ],
            "routes": {
                "": "index",
                "contact": "contact"
            }
        }
    }
}
```

Every JavaScript file defined in a module will have a `module` object in scope which contains the `name` and `routes` for the module. In `js/routers/pages.js` we could define a Backbone router for our `pages` module like so:

```
new (Backbone.Router.extend({
    routes: module.routes,
    index: function() {},
    contact: function() {}
}));
```

Module loader Router

A little used feature of `Backbone.Router` is it's ability to create multiple routers that listen to the same set of routes. Lumbar uses this feature to create a router that listens to all routes in the application. When a route is matched, this master router checks to see if the needed module is loaded. If the module is already loaded, then the master router takes no action and the router defined by the module will handle the route. If the needed module has not yet been loaded, it will be loaded, then `Backbone.history.loadUrl` will be called. This reloads the route, causes the master router to take no further action and the router defined in the freshly loaded module to respond.

A sample implementation is provided below. The `config` object would need to contain the data from our sample configuration JSON file above, and the `loader` object would need to implement `isLoaded` and `loadModule` methods. Note that Lumbar provides all of these implementations, the examples are provided to create your own implementation.

```
// Create an object that will be used as the prototype
// for our master router
var handlers = {
    routes: {}
};

_.each(config.modules, function(module, moduleName) {
    if (module.routes) {
        // Generate a loading callback for the module
        var callbackName = "loader_" moduleName;
        handlers[callbackName] = function() {
            if (loader.isLoaded(moduleName)) {
                // Do nothing if the module is loaded
                return;
            } else {
                //the module needs to be loaded
                loader.loadModule(moduleName, function() {
                    // Module is loaded, reloading the route
                    // will trigger callback in the module's
                    // router
                    Backbone.history.loadUrl();
                });
            }
        };
        // Each route in the module should trigger the
        // loading callback
        _.each(module.routes, function(methodName, route) {
            handlers.routes[route] = callbackName;
        });
    }
});

// Create the master router
new (Backbone.Router.extend(handlers));
```

Using NodeJS to handle pushState

`window.history.pushState` support (serving Backbone routes without a hash-tag) requires that the server be aware of what URLs your Backbone application will handle, since the user should be able to enter the app at any of those routes (or hit reload after navigating to a pushState URL).

Another advantage to defining all routes in a single location is that the same JSON configuration file provided above could be loaded by the server, listening to each route. A sample implementation in NodeJS and Express:

```
var fs = require('fs'),
    _ = require('underscore'),
    express = require('express'),
    server = express.createServer(),
    config = JSON.parse(fs.readFileSync('path/to/config.json'));

_.each(config.modules, function(module, moduleName) {
    if (module.routes) {
        _.each(module.routes, function(methodName, route) {
            server.get(route, function(req, res) {
                  res.sendFile('public/index.html');
            });
        });
    }
});
```

This assumes that index.html will be serving out your Backbone application. The `Backbone.History` object can handle the rest of the routing logic as long as a `root` option is specified. A sample configuration for a simple application that lives at the root might look like:

```
Backbone.history || (Backbone.history = new Backbone.History());
Backbone.history.start({
  pushState: true,
  root: '/'
});
```

Decoupling Backbone with the Mediator and Facade Patterns

In this section we'll discuss applying some of the concepts I cover in my article on Large-scale JavaScript Application development to Backbone.

After, you may be interested in taking a look At Aura - my popular widget-based Backbone.js extension framework based on many of the concepts we will be covering in this section.

Summary

At a high-level, one architecture that works for such applications is something which is:

- **Highly decoupled**: encouraging modules to only publish and subscribe to events of interest rather than directly communicating with each other. This helps us to build applications who's units of code aren't highly tied (coupled) together and can thus be reused more easily.
- **Supports module-level security**: whereby modules are only able to execute behavior they've been permitted to. Application security is an area which is often overlooked in JavaScript applications, but can be quite easily implemented in a flexible manner.
- **Supports failover**: allowing an application continuing to function even if particular modules fail. The typical example I give of this is the GMail chat widget. Imagine being able to build applications in a way that if one widget on the page fails (e.g chat), the rest of your application (mail) can continue to function without being affected.

This is an architecture which has been implemented by a number of different companies in the past, including Yahoo! (for their modularized homepage.

The three design patterns that make this architecture possible are the:

- **Module pattern**: used for encapsulating unique blocks of code, where functions and variables can be kept either public or private. ('private' in the simulation of privacy sense, as of course don't have true privacy in JavaScript)
- **Mediator pattern**: used when the communication between modules may be complex, but is still well defined. If it appears a system may have too many relationships between modules in your code, it may be time to have a central point of control, which is where the pattern fits in.
- **Facade pattern**: used for providing a convenient higher-level interface to a larger body of code, hiding its true underlying complexity

Their specific roles in this architecture can be found below.

- **Modules**: There are almost two concepts of what defines a module. As AMD is being used as a module wrapper, technically each model, view and collection can be considered a module. We then have the concept of modules being distinct blocks of code outside of just MVC/MV*. For the latter, these types of 'modules' are primarily concerned with broadcasting and subscribing to events of interest rather than directly communicating with each other.They are made possible through the Mediator pattern.

- **Mediator**: The mediator has a varying role depending on just how you wish to implement it. In my article, I mention using it as a module manager with the ability to start and stop modules at will, however when it comes to Backbone, I feel that simplifying it down to the role of a central 'controller' that provides pub/sub capabilities should suffice. One can of course go all out in terms of building a module system that supports module starting, stopping, pausing etc, however the scope of this is outside of this chapter.
- **Facade**: This acts as a secure middle-layer that both abstracts an application core (Mediator) and relays messages from the modules back to the Mediator so they don't touch it directly. The Facade also performs the duty of application security guard; it checks event notifications from modules against a configuration (permissions.js, which we will look at later) to ensure requests from modules are only processed if they are permitted to execute the behavior passed.

Exercise

For the practical section of this chapter, we'll be extending the well-known Backbone Todo application using the three patterns mentioned above.

The application is broken down into AMD modules that cover everything from Backbone models through to application-level modules. The views publish events of interest to the rest of the application and modules can then subscribe to these event notifications.

All subscriptions from modules go through a facade (or sandbox). What this does is check against the subscriber name and the 'channel/notification' it's attempting to subscribe to. If a channel *doesn't* have permissions to be subscribed to (something established through permissions.js), the subscription isn't permitted.

Mediator

Found in `aura/mediator.js`

Below is a very simple AMD-wrapped implementation of the mediator pattern, based on prior work by Ryan Florence. It accepts as its input an object, to which it attaches `publish()` and `subscribe()` methods. In a larger application, the mediator can contain additional utilities, such as handlers for initializing, starting and stopping modules, but for demonstration purposes, these two methods should work fine for our needs.

```
define([], function(obj){
  var channels = {};
  if (!obj) obj = {};

  obj.subscribe = function (channel, subscription) {
    if (!channels[channel]) channels[channel] = [];
    channels[channel].push(subscription);
  };

  obj.publish = function (channel) {
    if (!channels[channel]) return;
    var args = [].slice.call(arguments, 1);
    for (var i = 0, l = channels[channel].length; i < l; i++) {
      channels[channel][i].apply(this, args);
    }
  };

  return obj;

});
```

Facade

Found in `aura/facade.js`

Next, we have an implementation of the facade pattern. Now the classical facade pattern applied to JavaScript would probably look a little like this:

```
var module = (function() {
    var _private = {
        i:5,
        get : function() {
            console.log('current value:' + this.i);
        },
        set : function( val ) {
            this.i = val;
        },
        run : function() {
            console.log('running');
        },
        jump: function(){
            console.log('jumping');
        }
    };
    return {
        facade : function( args ) {
            _private.set(args.val);
            _private.get();
            if ( args.run ) {
                _private.run();
            }
        }
    }
}());

module.facade({run: true, val:10});
//outputs current value: 10, running
```

It's effectively a variation of the module pattern, where instead of simply returning an interface of supported methods, your API can completely hide the true implementation powering it, returning something simpler. This allows the logic being performed in the background to be as complex as necessary, whilst all the end-user experiences is a simplified API they pass options to (note how in our case, a single method abstraction is exposed). This is a beautiful way of providing APIs that can be easily consumed.

That said, to keep things simple, our implementation of an AMD-compatible facade will act a little more like a proxy. Modules will communicate directly through the facade to access the mediator's `publish()` and `subscribe()` methods, however, they won't as such touch the mediator directly.This enables the facade to provide application-level validation of any subscriptions and publications made.

It also allows us to implement a simple, but flexible, permissions checker (as seen below) which will validate subscriptions made against a permissions configuration to see whether it's permitted or not.

```
define([ '../aura/mediator' , '../aura/permissions' ], function (mediator, permissions) {

    var facade = facade || {};

    facade.subscribe = function(subscriber, channel, callback){

        // Note: Handling permissions/security is optional here
        // The permissions check can be removed
        // to just use the mediator directly.

        if(permissions.validate(subscriber, channel)){
            mediator.subscribe( channel, callback );
        }
    }
    facade.publish = function(channel){
        mediator.publish( channel );
    }
    return facade;

});
```

Permissions

Found in `aura/permissions.js`

In our simple permissions configuration, we support checking against subscription requests to establish whether they are allowed to clear. This enforces a flexible security layer for the application.

To visually see how this works, consider changing say, permissions -> renderDone -> todoCounter to be false. This will completely disable the application from from rendering or displaying the counts component for Todo items left (because they aren't allowed to subscribe to that event notification). The rest of the Todo app can still however be used without issue.

It's a very dumbed down example of the potential for application security, but imagine how powerful this might be in a large app with a significant number of visual widgets.

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

    // Permissions

    // A permissions structure can support checking
    // against subscriptions prior to allowing them
    // to clear. This enforces a flexible security
    // layer for your application.

    var permissions = {

        newContentAvailable: {
            contentUpdater:true
        },

        endContentEditing:{
            todoSaver:true
        },

        beginContentEditing:{
            editFocus:true
        },
        addingNewTodo:{
            todoTooltip:true
        },

        clearContent:{
            garbageCollector:true
        },

        renderDone:{
            todoCounter:true //switch to false to see what happens :)
        },

        destroyContent:{
            todoRemover:true
        },

        createWhenEntered:{
            keyboardManager:true
        }

    };

    permissions.validate = function(subscriber, channel){
        var test = permissions[channel][subscriber];
        return test===undefined? false: test;
    };

    return permissions;

});
```

Subscribers

Found in `subscribers.js`

Subscriber 'modules' communicate through the facade back to the mediator and perform actions when a notification event of a particular name is published.

For example, when a user enters in a new piece of text for a Todo item and hits 'enter' the application publishes a notification saying two things: a) a new Todo item is available and b) the text content of the new item is X. It's then left up to the rest of the application to do with this information whatever it wishes.

In order to update your Backbone application to primarily use pub/sub, a lot of the work you may end up doing will be moving logic coupled inside of specific views to modules outside of it which are reactionary.

Take the todoSaver for example - its responsibility is saving new Todo items to models once the a notificationName called 'newContentAvailable' has fired. If you take a look at the permissions structure in the last code sample, you'll notice that 'newContentAvailable' is present there. If I wanted to prevent subscribers from being able to subscribe to this notification, I simply set it to a boolean value of false.

Again, this is a massive oversimplification of how advanced your permissions structures could get, but it's certainly one way of controlling what parts of your application can or can't be accessed by specific modules at any time.

```
define(['jquery', 'underscore', 'aura/facade'],
function ($, _, facade) {

    // Subscription 'modules' for our views. These take the
    // the form facade.subscribe( subscriberName, notificationName , callBack )

    // Update view with latest todo content
    // Subscribes to: newContentAvailable

    facade.subscribe('contentUpdater', 'newContentAvailable', function (context) {
        var content = context.model.get('content');
        context.$('.todo-content').text(content);
        context.input = context.$('.todo-input');
        context.input.bind('blur', context.close);
        context.input.val(content);
    });


    // Save models when a user has finishes editing
    // Subscribes to: endContentEditing
    facade.subscribe('todoSaver','endContentEditing', function (context) {
        try {
            context.model.save({
                content: context.input.val()
            });
            context.$el.removeClass('editing');
        } catch (e) {
            //console.log(e);
        }
    });


    // Delete a todo when the user no longer needs it
    // Subscribes to: destroyContent
    facade.subscribe('todoRemover','destroyContent', function (context) {
```

```
    try {
        context.model.clear();
    } catch (e) {
        //console.log(e);
    }
});


// When a user is adding a new entry, display a tooltip
// Subscribes to: addingNewTodo
facade.subscribe('todoTooltip','addingNewTodo', function (context, todo) {
    var tooltip = context.$('.ui-tooltip-top');
    var val = context.input.val();
    tooltip.fadeOut();
    if (context.tooltipTimeout) clearTimeout(context.tooltipTimeout);
    if (val == '' || val == context.input.attr('placeholder')) return;
    var show = function () {
            tooltip.show().fadeIn();
        };
    context.tooltipTimeout = _.delay(show, 1000);
});


// Update editing UI on switching mode to editing content
// Subscribes to: beginContentEditing
facade.subscribe('editFocus','beginContentEditing', function (context) {
    context.$el.addClass('editing');
    context.input.focus();
});


// Create a new todo entry
// Subscribes to: createWhenEntered
facade.subscribe('keyboardManager','createWhenEntered', function (context, e, todos) {
    if (e.keyCode != 13) return;
    todos.create(context.newAttributes());
    context.input.val('');
});



// A Todo and remaining entry counter
// Subscribes to: renderDone
facade.subscribe('todoCounter','renderDone', function (context, Todos) {
    var done = Todos.done().length;
    context.$('#todo-stats').html(context.statsTemplate({
            total: Todos.length,
            done: Todos.done().length,
            remaining: Todos.remaining().length
        }));
    });


    // Clear all completed todos when clearContent is dispatched
    // Subscribes to: clearContent
    facade.subscribe('garbageCollector','clearContent', function (Todos) {
        _.each(Todos.done(), function (todo) {
            todo.clear();
        });
    });


});
```

That's it for this section. If you've been intrigued by some of the concepts covered, I encourage you to consider taking a look at my slides on Large-scale JS from the jQuery Summit or my longer post on the topic here for more information.

Paginating Backbone.js Requests & Collections

Pagination is a ubiquitous problem we often find ourselves needing to solve on the web. Perhaps most predominantly when working with back-end APIs and JavaScript-heavy clients which consume them.

On this topic, we're going to go through a set of **pagination components** I wrote for Backbone.js, which should hopefully come in useful if you're working on applications which need to tackle this problem. They're part of an extension called Backbone.Paginator.

When working with a structural framework like Backbone.js, the three types of pagination we are most likely to run into are:

Requests to a service layer (API)- e.g query for results containing the term 'Brendan' - if 5,000 results are available only display 20 results per page (leaving us with 250 possible result pages that can be navigated to).

This problem actually has quite a great deal more to it, such as maintaining persistence of other URL parameters (e.g sort, query, order) which can change based on a user's search configuration in a UI. One also had to think of a clean way of hooking views up to this pagination so you can easily navigate between pages (e.g First, Last, Next, Previous, 1,2,3), manage the number of results displayed per page and so on.

Further client-side pagination of data returned - e.g we've been returned a JSON response containing 100 results. Rather than displaying all 100 to the user, we only display 20 of these results within a navigatable UI in the browser.

Similar to the request problem, client-pagination has its own challenges like navigation once again (Next, Previous, 1,2,3), sorting, order, switching the number of results to display per page and so on.

Infinite results - with services such as Facebook, the concept of numeric pagination is instead replaced with a 'Load More' or 'View More' button. Triggering this normally fetches the next 'page' of N results but rather than replacing the previous set of results loaded entirely, we simply append to them instead.

A request pager which simply appends results in a view rather than replacing on each new fetch is effectively an 'infinite' pager.

Let's now take a look at exactly what we're getting out of the box:

Paginator is a set of opinionated components for paginating collections of data using Backbone.js. It aims to provide both solutions for assisting with pagination of requests to a server (e.g an API) as well as pagination of single-loads of data, where we may wish to further paginate a collection of N results into M pages within a view.

Paginator's pieces

Backbone.Paginator supports two main pagination components:

- **Backbone.Paginator.requestPager**: For pagination of requests between a client and a server-side API
- **Backbone.Paginator.clientPager**: For pagination of data returned from a server which you would like to further paginate within the UI (e.g 60 results are returned, paginate into 3 pages of 20)

Live Examples

Live previews of both pagination components using the Netflix API can be found below. Fork the repository to experiment with these examples further.

- Backbone.Paginator.requestPager()
- Backbone.Paginator.clientPager()
- Infinite Pagination (Backbone.Paginator.requestPager())
- Diacritic Plugin

Paginator.requestPager

In this section we're going to walkthrough actually using the requestPager.

1. Create a new Paginated collection First, we define a new Paginated collection using `Backbone.Paginator.requestPager()` as follows:

```
var PaginatedCollection = Backbone.Paginator.requestPager.extend({
```

2: Set the model for the collection as normal Within our collection, we then (as normal) specify the model to be used with this collection followed by the URL (or base URL) for the service providing our data (e.g the Netflix API).

```
        model: model,
```

3. Configure the base URL and the type of the request We need to set a base URL. The `type` of the request is `GET` by default, and the `dataType` is `jsonp` in order to enable cross-domain requests.

```
    paginator_core: {
      // the type of the request (GET by default)
      type: 'GET',

      // the type of reply (jsonp by default)
      dataType: 'jsonp',

      // the URL (or base URL) for the service
      url: 'http://odata.netflix.com/Catalog/People(49446)/TitlesActedIn?'
    },
```

4. Configure how the library will show the results We need to tell the library how many items per page would we like to see, etc…

```
    paginator_ui: {
      // the lowest page index your API allows to be accessed
      firstPage: 0,

      // which page should the paginator start from
      // (also, the actual page the paginator is on)
      currentPage: 0,

      // how many items per page should be shown
      perPage: 3,

      // a default number of total pages to query in case the API or
      // service you are using does not support providing the total
      // number of pages for us.
      // 10 as a default in case your service doesn't return the total
      totalPages: 10
    },
```

5. Configure the parameters we want to send to the server Only the base URL won't be enough for most cases, so you can pass more parameters to the server. Note how you can use functions insead of hardcoded values, and you can also reffer to the values you specified in `paginator_ui`.

```
  server_api: {
    // the query field in the request
    '$filter': '',

    // number of items to return per request/page
    '$top': function() { return this.perPage },

    // how many results the request should skip ahead to
    // customize as needed. For the Netflix API, skipping ahead based on
    // page * number of results per page was necessary.
    '$skip': function() { return this.currentPage * this.perPage },

    // field to sort by
    '$orderby': 'ReleaseYear',

    // what format would you like to request results in?
    '$format': 'json',

    // custom parameters
    '$inlinecount': 'allpages',
    '$callback': 'callback'
  },
```

6. Finally, configure Collection.parse() and we're done The last thing we need to do is configure our collection's `parse()` method. We want to ensure we're returning the correct part of our JSON response containing the data our collection will be populated with, which below is `response.d.results` (for the Netflix API).

You might also notice that we're setting `this.totalPages` to the total page count returned by the API. This allows us to define the maximum number of (result) pages available for the current/last request so that we can clearly display this in the UI. It also allows us to infuence whether clicking say, a 'next' button should proceed with a request or not.

```
        parse: function (response) {
            // Be sure to change this based on how your results
            // are structured (e.g d.results is Netflix specific)
            var tags = response.d.results;
            //Normally this.totalPages would equal response.d.__count
            //but as this particular NetFlix request only returns a
            //total count of items for the search, we divide.
            this.totalPages = Math.floor(response.d.__count / this.perPage);
            return tags;
        }
    });

});
```

Convenience methods: For your convenience, the following methods are made available for use in your views to interact with the `requestPager`:

- **Collection.goTo(n, options)** - go to a specific page
- **Collection.requestNextPage(options)** - go to the next page
- **Collection.requestPreviousPage(options)** - go to the previous page
- **Collection.howManyPer(n)** - set the number of items to display per page

requestPager collection's methods `.goTo()`, `.requestNextPage()` and `.requestPreviousPage()` are all extension of the original Backbone Collection.fetch() method. As so, they all can take the same option object as parameter.

This option object can use `success` and `error` parameters to pass a function to be executed after server answer.

```
Collection.goTo(n, {
  success: function( collection, response ) {
    // called is server request success
  },
  error: function( collection, response ) {
    // called if server request fail
  }
});
```

To manage callback, you could also use the jqXHR returned by these methods to manage callback.

```
Collection
  .requestNextPage()
  .done(function( data, textStatus, jqXHR ) {
    // called is server request success
  })
  .fail(function( data, textStatus, jqXHR ) {
    // called if server request fail
  })
  .always(function( data, textStatus, jqXHR ) {
    // do something after server request is complete
  });
});
```

If you'd like to add the incoming models to the current collection, instead of replacing the collection's contents, pass `{add: true}` as an option to these methods.

```
Collection.requestPreviousPage({ add: true });
```

Paginator.clientPager

The `clientPager` works similar to the `requestPager`, except that our configuration values influence the pagination of data already returned at a UI-level. Whilst not shown (yet) there is also a lot more UI logic that ties in with the `clientPager`. An example of this can be seen in 'views/clientPagination.js'.

1. Create a new paginated collection with a model and URL As with `requestPager`, let's first create a new Paginated `Backbone.Paginator.clientPager` collection, with a model:

```
    var PaginatedCollection = Backbone.Paginator.clientPager.extend({

        model: model,
```

2. Configure the base URL and the type of the request We need to set a base URL. The `type` of the request is `GET` by default, and the `dataType` is `jsonp` in order to enable cross-domain requests.

```
    paginator_core: {
      // the type of the request (GET by default)
      type: 'GET',

      // the type of reply (jsonp by default)
      dataType: 'jsonp',
      // the URL (or base URL) for the service
      url: 'http://odata.netflix.com/v2/Catalog/Titles?&'
    },
```

3. Configure how the library will show the results We need to tell the library how many items per page would we like to see, etc. . .

```
  paginator_ui: {
    // the lowest page index your API allows to be accessed
    firstPage: 1,

    // which page should the paginator start from
    // (also, the actual page the paginator is on)
    currentPage: 1,

    // how many items per page should be shown
    perPage: 3,

    // a default number of total pages to query in case the API or
    // service you are using does not support providing the total
    // number of pages for us.
    // 10 as a default in case your service doesn't return the total
    totalPages: 10
  },
```

4. Configure the parameters we want to send to the server Only the base URL won't be enough for most cases, so you can pass more parameters to the server. Note how you can use functions insead of hardcoded values, and you can also reffer to the values you specified in `paginator_ui`.

```
  server_api: {
    // the query field in the request
    '$filter': 'substringof(\'america\',Name)',

    // number of items to return per request/page
    '$top': function() { return this.perPage },

    // how many results the request should skip ahead to
    // customize as needed. For the Netflix API, skipping ahead based on
    // page * number of results per page was necessary.
    '$skip': function() { return this.currentPage * this.perPage },

    // field to sort by
    '$orderby': 'ReleaseYear',
    // what format would you like to request results in?
    '$format': 'json',

    // custom parameters
    '$inlinecount': 'allpages',
    '$callback': 'callback'
  },
```

5. Finally, configure Collection.parse() and we're done And finally we have our `parse()` method, which in this case isn't concerned with the total number of result pages available on the server as we have our own total count of pages for the paginated data in the UI.

```
    parse: function (response) {
            var tags = response.d.results;
            return tags;
        }

    });
```

Convenience methods: As mentioned, your views can hook into a number of convenience methods to navigate around UI-paginated data. For `clientPager` these include:

- **Collection.goTo(n)** - go to a specific page
- **Collection.previousPage()** - go to the previous page
- **Collection.nextPage()** - go to the next page
- **Collection.howManyPer(n)** - set how many items to display per page
- **Collection.setSort(sortBy, sortDirection)** - update sort on the current view. Sorting will automatically detect if you're trying to sort numbers (even if they're strored as strings) and will do the right thing.
- **Collection.setFilter(filterFields, filterWords)** - filter the current view. Filtering supports multiple words without any specific order, so you'll basically get a full-text search ability. Also, you can pass it only one field from the model, or you can pass an array with fields and all of them will get filtered. Last option is to pass it an object containing a comparison method and rules. Currently, only `levenshtein` method is available.

```
  this.collection.setFilter(
    {'Name': {cmp_method: 'levenshtein', max_distance: 7}}
    , 'Amreican P' // Note the switched 'r' and 'e', and the 'P' from 'Pie'
  );
```

Also note that the levenshtein plugin should be loaded and enabled using the `useLevenshteinPlugin` variable.

Last but not less important: Performing Levenshtein comparison returns the `distance` between to strings. It won't let you *search* lenghty text.

The distance between two strings means the number of characters that should be added, removed or moved to the left or to the right so the strings get equal.

That means that comparing "Something" in "This is a test that could show something" will return 32, which is bigger than comparing "Something" and "ABCDEFG" (9).

Use levenshtein only for short texts (titles, names, etc).

- **Collection.doFakeFilter(filterFields, filterWords)** - returns the models count after fake-applying a call to `Collection.setFilter`.

- **Collection.setFieldFilter(rules)** - filter each value of each model according to `rules` that you pass as argument. Example: You have a collection of books with 'release year' and 'author'. You can filter only the books that were released between 1999 and 2003. And then you can add another `rule` that will filter those books only to authors who's name start with 'A'. Possible rules: function, required, min, max, range, minLength, maxLength, rangeLength, oneOf, equalTo, pattern.

```
my_collection.setFieldFilter([
  {field: 'release_year', type: 'range', value: {min: '1999', max: '2003'}},
  {field: 'author', type: 'pattern', value: new RegExp('A*', 'igm')}
]);

//Rules:
//
//var my_var = 'green';
//
//{field: 'color', type: 'equalTo', value: my_var}
//{field: 'color', type: 'function', value: function(field_value){ return field_value ==
//{field: 'color', type: 'required'}
//{field: 'number_of_colors', type: 'min', value: '2'}
//{field: 'number_of_colors', type: 'max', value: '4'}
//{field: 'number_of_colors', type: 'range', value: {min: '2', max: '4'} }
//{field: 'color_name', type: 'minLength', value: '4'}
//{field: 'color_name', type: 'maxLength', value: '6'}
//{field: 'color_name', type: 'rangeLength', value: {min: '4', max: '6'}}
//{field: 'color_name', type: 'oneOf', value: ['green', 'yellow']}
//{field: 'color_name', type: 'pattern', value: new RegExp('gre*', 'ig')}
```

- **Collection.doFakeFieldFilter(rules)** - returns the models count after fake-applying a call to `Collection.setFieldFilter`.

Implementation notes: You can use some variables in your `View` to represent the actual state of the paginator.

`totalUnfilteredRecords` - Contains the number of records, including all records filtered in any way. (Only available in `clientPager`)

`totalRecords` - Contains the number of records

`currentPage` - The actual page were the paginator is at.

`perPage` - The number of records the paginator will show per page.

`totalPages` - The number of total pages.

`startRecord` - The posicion of the first record shown in the current page (eg 41 to 50 from 2000 records) (Only available in `clientPager`)

`endRecord` - The posicion of the last record shown in the current page (eg 41 to 50 from 2000 records) (Only available in `clientPager`)

Plugins

Diacritic.js

A plugin for Backbone.Paginator that replaces diacritic characters (ă,ş,ţ etc) with characters that match them most closely. This is particularly useful for filtering.

To enable the plugin, set `this.useDiacriticsPlugin` to true, as can be seen in the example below:

```
Paginator.clientPager = Backbone.Collection.extend({

    // Default values used when sorting and/or filtering.
    initialize: function(){
      this.useDiacriticsPlugin = true; // use diacritics plugin if available
    ...
```

Thorax

By Ryan Eastridge & Addy Osmani

view helper

Creating new View helpers

Note that this differs from `Handlebars.registerHelper`. Registers a helper that will create and append a new `HelperView` instance, with it's `template` attribute set to the value of the captured block. `callback` will recieve any arguments passed to the helper followed by a `HelperView` instance. Named arguments to the helper will be present on `options` attribute of the `HelperView` instance.

A `HelperView` instance differs from a regular view instance in that it has a `parent` attribute which is always set to the declaring view, and a `context` which always returns the value of the `parent`'s context method. The `collection`, `empty` and other built in block view helpers are created with `registerViewHelper`.

A helper that re-rendered a `HelperView` every time an event was triggered on the declaring / parent view could be implemented as:

```
Handlebars.registerViewHelper('on', function(eventName, helperView) {
  // register a handler on the parent view, which will be automatically
  // unregistered when helperView is destroyed
  helperView.on(helperView.parent, eventName, function() {
    helperView.render();
  });
});
```

An example use of this would be to have a counter that would incriment each time a button was clicked. In Handlebars:

```
{{#on "incrimented"}}{{i}}{/on}}
{{#button trigger="incrimented"}}Add{{/button}}
```

And the corresponding view class:

```
new Thorax.View({
    events: {
        incrimented: function() {
            ++this.i;
        }
    },
    initialize: function() {
        this.i = 0;
    },
    template: ...
});
```

- *.view*,.model, $.collection

- registry
- template helper
- straight embed
- yield
- view helper
- registerViewHelper
- collection helper
- layout and view lifecycle
- bindToRoute
- mobile

Mobile Applications

Backbone & jQuery Mobile

Resolving the routing conflicts

The first major hurdle developers typically run into when building Backbone applications with jQuery Mobile is that both frameworks have their own opinions about how to handle application navigation.

Backbone's routers offer an explicit way to define custom navigation routes through `Backbone.Router`, whilst jQuery Mobile encourages the use of URL hash fragments to reference separate 'pages' or views in the same document. jQuery Mobile also supports automatically pulling in external content for links through XHR calls meaning that there can be quite a lot of inter-framework confusion about what a link pointing at '#photo/id' should actually be doing.

Some of the solutions that have been previously proposed to work-around this problem included manually patching Backbone or jQuery Mobile. I discourage opting for these techniques as it becomes necessary to manually patch your framework builds when new releases get made upstream.

There's also jQueryMobile router, which tries to solve this problem differently, however I think my proposed solution is both simpler and allows both frameworks to cohabit quite peacefully without the need to extend either. What we're after is a way to prevent one framework from listening to hash changes so that we can fully rely on the other (e.g. `Backbone.Router`) to handle this for us exclusively.

Using jQuery Mobile this can be done by setting:

```
$.mobile.hashListeningEnabled = false;
```

prior to initializing any of your other code.

I discovered this method looking through some jQuery Mobile commits that didn't make their way into the official docs, but am happy to see that they are now covered here http://jquerymobile.com/test/docs/api/globalconfig.html in more detail.

The next question that arises is, if we're preventing jQuery Mobile from listening to URL hash changes, how can we still get the benefit of being able to navigate to other sections in a document using the built-in transitions and effects supported? Good question. This can now be solve by simply calling `$.mobile.changePage()` as follows:

```
var url = '#about',
    effect = 'slideup',
    reverse = false,
    changeHash = false;

$.mobile.changePage( url , { transition: effect}, reverse, changeHash );
```

In the above sample, `url` can refer to a URL or a hash identifier to navigate to, `effect` is simply the transition effect to animate the page in with and the final two parameters decide the direction for the transition (`reverse`) and whether or not the hash in the address bar should be updated (`changeHash`). With respect to the latter, I typically set this to false to avoid managing two sources for hash updates, but feel free to set this to true if you're comfortable doing so.

Note: For some parallel work being done to explore how well the jQuery Mobile Router plugin works with Backbone, you may be interested in checking out https://github.com/Filirom1/jquery-mobile-backbone-requirejs.

Exercise: A Backbone, Require.js/AMD app with jQuery Mobile

Note: The code for this exercise can be found in `practicals/modular-mobile-app`.

Getting started

Once you feel comfortable with the Backbone fundamentals and you've put together a rough wireframe of the app you may wish to build, start to think about your application architecture. Ideally, you'll want to logically separate concerns so that it's as easy as possible to maintain the app in the future.

Namespacing

For this application, I opted for the nested namespacing pattern. Implemented correctly, this enables you to clearly identify if items being referenced in your app are views, other modules and so on. This initial structure is a sane place to also include application defaults (unless you prefer maintaining those in a separate file).

```
window.mobileSearch = window.mobileSearch || {
    views: {
        appview: new AppView()
    },
    routers:{
        workspace:new Workspace()
    },
    utils: utils,
    defaults: {
        resultsPerPage: 16,
        safeSearch: 2,
        maxDate:'',
        minDate:'01/01/1970'
    }
}
```

Models

In the Flickly application, there are at least two unique types of data that need to be modeled - search results and individual photos, both of which contain additional meta-data like photo titles. If you simplify this down, search results are actually groups of photos in their own right, so the application only requires:

- A single model (a photo or 'result' entry)
- A result collection (containing a group of result entries) for search results
- A photo collection (containing one or more result entries) for individual photos or photos with more than one image

Views

The views we'll need include an application view, a search results view and a photo view. Static views or pages of the single-page application which do not require a dynamic element to them (e.g an 'about' page) can be easily coded up in your document's markup, independent of Backbone.

Routers

A number of possible routes need to be taken into consideration:

- Basic search queries `#search/kiwis`

- Search queries with additional parameters (e.g sort, pagination) `#search/kiwis/srelevance/p7`
- Queries for specific photos `#photo/93839`
- A default route (no parameters passed)

This tutorial will be expanded shortly to fully cover the demo application. In the mean time, please see the practicals folder for the completed application that demonstrates the router resolution discussed earlier between Backbone and jQuery Mobile.

jQuery Mobile: Going beyond mobile application development

The majority of jQM apps I've seen in production have been developed for the purpose of providing an optimal experience to users on mobile devices. Given that the framework was developed for this purpose, there's nothing fundamentally wrong with this, but many developers forget that jQM is a UI framework not dissimilar to jQuery UI. It's using the widget factory and is capable of being used for a lot more than we give it credit for.

If you open up Flickly in a desktop browser, you'll get an image search UI that's modeled on Google.com, however, review the components (buttons, text inputs, tabs) on the page for a moment. The desktop UI doesn't look anything like a mobile application yet I'm still using jQM for theming mobile components; the tabs, date-picker, sliders - everything in the desktop UI is re-using what jQM would be providing users on mobile devices. Thanks to some media queries, the desktop UI can make optimal use of whitespace, expanding component blocks out and providing alternative layouts whilst still making use of jQM as a component framework.

The benefit of this is that I don't need to go pulling in jQuery UI separately to be able to take advantage of these features. Thanks to the recent ThemeRoller my components can look pretty much exactly how I would like them to and users of the app can get a jQM UI for lower-resolutions and a jQM-ish UI for everything else.

The takeaway here is just to remember that if you're not (already) going through the hassle of conditional script/style loading based on screen-resolution (using matchMedia.js etc), there are simpler approaches that can be taken to cross-device component theming.

Unit Testing

Jasmine

Introduction

One definition of unit testing is the process of taking the smallest piece of testable code in an application, isolating it from the remainder of your codebase and determining if it behaves exactly as expected. In this section, we'll be taking a look at how to unit test Backbone applications using a popular JavaScript testing framework called Jasmine from Pivotal Labs.

For an application to be considered 'well'-tested, distinct functionality should ideally have its own separate unit tests where it's tested against the different conditions you expect it to work under. All tests must pass before functionality is considered 'complete'. This allows developers to both modify a unit of code and its dependencies with a level of confidence about whether these changes have caused any breakage.

As a basic example of unit testing is where a developer may wish to assert whether passing specific values through to a sum function results in the correct output being returned. For an example more relevant to this book, we may wish to assert whether a user adding a new Todo item to a list correctly adds a Model of a specific type to a Todos Collection.

When building modern web-applications, it's typically considered best-practice to include automated unit testing as a part of your development process. Whilst we'll be focusing on Jasmine as a solution for this, there are a number of other alternatives worth considering, including QUnit.

Jasmine describes itself as a behavior-driven development (BDD) framework for testing JavaScript code. Before we jump into how the framework works, it's useful to understand exactly what BDD is.

BDD is a second-generation testing approach first described by Dan North (the authority on BDD) which attempts to test the behavior of software. It's considered second-generation as it came out of merging ideas from Domain driven design (DDD) and lean software development, helping teams to deliver high quality software by answering many of the more confusing questions early on in the agile process. Such questions commonly include those concerning documentation and testing.

If you were to read a book on BDD, it's likely to also be described as being 'outside-in and pull-based'. The reason for this is that it borrows the idea of pulling features from Lean manufacturing which effectively ensures that the right software solutions are being written by a) focusing on expected outputs of the system and b) ensuring these outputs are achieved.

BDD recognizes that there are usually multiple stakeholders in a project and not a single amorphous user of the system. These different groups will be affected by the software being written in differing ways and will have a varying opinion of what quality in the system means to them. It's for this reason that it's important to understand who the software will be bringing value you and exactly what in it will be valuable to them.

Finally, BDD relies on automation. Once you've defined the quality expected, your team will likely want to check on the functionality of the solution being built regularly and compare it to the results they expect. In order to facilitate this efficiently, the process has to be automated. BDD relies heavily on the automation of specification-testing and Jasmine is a tool which can assist with this.

BDD helps both developers and non-technical stakeholders:

- Better understand and represent the models of the problems being solved
- Explain supported tests cases in a language that non-developers can read
- Focus on minimizing translation of the technical code being written and the domain language spoken by the business

What this means is that developers should be able to show Jasmine unit tests to a project stakeholder and (at a high level, thanks to a common vocabulary being used) they'll ideally be able to understand what the code supports.

Developers often implement BDD in unison with another testing paradigm known as TDD (test-driven development). The main idea behind TDD is:

- Write unit tests which describe the functionality you would like your code to support
- Watch these tests fail (as the code to support them hasn't yet been written)
- Write code to make the tests pass
- Rinse, repeat and refactor

In this chapter we're going to use both BDD (with TDD) to write unit tests for a Backbone application.

Note: I've seen a lot of developers also opt for writing tests to validate behavior of their code after having written it. While this is fine, note that it can come with pitfalls such as only testing for behavior your code currently supports, rather than behavior the problem needs to be supported.

Suites, Specs & Spies

When using Jasmine, you'll be writing suites and specifications (specs). Suites basically describe scenarios whilst specs describe what can be done in these scenarios.

Each spec is a JavaScript function, described with a call to `it()` using a description string and a function. The description should describe the behaviour the particular unit of code should exhibit and keeping in mind BDD, it should ideally be meaningful. Here's an example of a basic spec:

```
it('should be incrementing in value', function(){
    var counter = 0;
    counter++;
});
```

On its own, a spec isn't particularly useful until expectations are set about the behavior of the code. Expectations in specs are defined using the `expect()` function and an expectation matcher (e.g `toEqual()`, `toBeTruthy()`, `toContain()`). A revised example using an expectation matcher would look like:

```
it('should be incrementing in value', function(){
    var counter = 0;
    counter++;
    expect(counter).toEqual(1);
});
```

The above code passes our behavioral expectation as `counter` equals 1. Notice how easy this was to read the expectation on the last line (you probably grokked it without any explanation).

Specs are grouped into suites which we describe using Jasmine's `describe()` function, again passing a string as a description and a function. The name/description for your suite is typically that of the component or module you're testing.

Jasmine will use it as the group name when it reports the results of the specs you've asked it to run. A simple suite containing our sample spec could look like:

```
describe('Stats', function(){
    it('can increment a number', function(){
        ...
    });

    it('can subtract a number', function(){
        ...
    });
});
```

Suites also share a functional scope and so it's possible to declare variables and functions inside a describe block which are accessible within specs:

```
describe('Stats', function(){
    var counter = 1;

    it('can increment a number', function(){
        // the counter was = 1
        counter = counter + 1;
        expect(counter).toEqual(2);
    });

    it('can subtract a number', function(){
        // the counter was = 2
        counter = counter - 1;
        expect(counter).toEqual(1);
    });
});
```

Note: Suites are executed in the order in which they are described, which can be useful to know if you would prefer to see test results for specific parts of your application reported first.

Jasmine also supports **spies** - a way to mock, spy and fake behavior in our unit tests. Spies replace the function they're spying on, allowing us to simulate behavior we would like to mock (i.e test free of the actual implementation).

In the below example, we're spying on the `setComplete` method of a dummy Todo function to test that arguments can be passed to it as expected.

```
var Todo = function(){
};

Todo.prototype.setComplete = function (arg){
    return arg;
}

describe('a simple spy', function(){
    it('should spy on an instance method of a Todo', function(){
        var myTodo = new Todo();
        spyOn(myTodo, 'setComplete');
        myTodo.setComplete('foo bar');

        expect(myTodo.setComplete).toHaveBeenCalledWith('foo bar');

        var myTodo2 = new Todo();
        spyOn(myTodo2, 'setComplete');

        expect(myTodo2.setComplete).not.toHaveBeenCalled();
    });
});
```

What you're more likely to use spies for is testing asynchronous behavior in your application such as AJAX requests. Jasmine supports:

- Writing tests which can mock AJAX requests using spies. This allows us to test code which runs before an AJAX request and right after. It's also possible to mock/fake responses the server can return and the benefit of this type of testing is that it's faster as no real calls are being made to a server
- Asynchronous tests which don't rely on spies

For the first kind of test, it's possible to both fake an AJAX request and verify that the request was both calling the correct URL and executed a callback where one was provided.

```
it('the callback should be executed on success', function () {
    spyOn($, 'ajax').andCallFake(function(options) {
        options.success();
    });

    var callback = jasmine.createSpy();
    getTodo(15, callback);

    expect($.ajax.mostRecentCall.args[0]['url']).toEqual('/todos/15');
    expect(callback).toHaveBeenCalled();
});

function getTodo(id, callback) {
    $.ajax({
        type: 'GET',
        url: '/todos/'' + id,
        dataType: 'json',
        success: callback
    });
}
```

If you feel lost having seen matchers like `andCallFake()` and `toHaveBeenCalled()`, don't worry. All of these are Spy-specific matchers and are documented on the Jasmine wiki.

For the second type of test (asynchronous tests), we can take the above further by taking advantage of three other methods Jasmine supports:

- runs(function) - a block which runs as if it was directly called
- waits(timeout) - a native timeout before the next block is run
- waitsFor(function, optional message, optional timeout) - a way to pause specs until some other work has completed. Jasmine waits until the supplied function returns true here before it moves on to the next block.

```
it('should make an actual AJAX request to a server', function () {

    var callback = jasmine.createSpy();
    getTodo(16, callback);

    waitsFor(function() {
        return callback.callCount > 0;
    });

    runs(function() {
        expect(callback).toHaveBeenCalled();
    });
});

function getTodo(id, callback) {
    $.ajax({
        type: 'GET',
        url: 'todos.json',
        dataType: 'json',
        success: callback
    });
}
```

Note: It's useful to remember that when making real requests to a web server in your unit tests, this has the potential to massively slow down the speed at which tests run (due to many factors including server latency). As this also introduces an external dependency that can (and should) be minimized in your unit testing, it is strongly recommended that you opt for spies to remove the need for a web server to be used here.

beforeEach and afterEach()

Jasmine also supports specifying code that can be run before each (`beforeEach()`) and after each (`afterEach`) test. This is useful for enforcing consistent conditions (such as resetting variables that may be required by specs). In the following example, `beforeEach()` is used to create a new sample Todo model specs can use for testing attributes.

```
beforeEach(function(){
   this.todo = new Backbone.Model({
      text: 'Buy some more groceries',
      done: false
   });
});

it('should contain a text value if not the default value', function(){
   expect(this.todo.get('text')).toEqual('Buy some more groceries');
});
```

Each nested `describe()` in your tests can have their own `beforeEach()` and `afterEach()` methods which support including setup and teardown methods relevant to a particular suite. We'll be using `beforeEach()` in practice a little later.

Shared scope

In the previous section you may have noticed that we initially declared a variable `this.todo` in our `beforeEach()` call and were then able to continue using this in `afterEach()`. This is thanks to a powerful feature of Jasmine known as shared functional scope. Shared scope allows `this` properties to be common to all blocks (including `runs()`), but not declared variables (i.e `var`s).

Getting setup

Now that we've reviewed some fundamentals, let's go through downloading Jasmine and getting everything setup to write tests.

A standalone release of Jasmine can be downloaded from the official release page.

You'll need a file called SpecRunner.html in addition to the release. It can be downloaded from https://github.com/pivotal/jasmine/tree/master/lib/jasmine-core/example or as part of a download of the complete Jasmine repo.Alternatively, you can `git clone` the main Jasmine repository from https://github.com/pivotal/jasmine.git.

Let's review SpecRunner.html:

It first includes both Jasmine and the necessary CSS required for reporting:

```
<link rel="stylesheet" type="text/css" href="lib/jasmine-1.1.0.rc1/jasmine.css"/>
<script type="text/javascript" src="lib/jasmine-1.1.0.rc1/jasmine.js"></script>
<script type="text/javascript" src="lib/jasmine-1.1.0.rc1/jasmine-html.js"></script>
```

Next, some sample tests are included:

```
<script type="text/javascript" src="spec/SpecHelper.js"></script>
<script type="text/javascript" src="spec/PlayerSpec.js"></script>
```

And finally the sources being tested:

```
<script type="text/javascript" src="src/Player.js"></script>
<script type="text/javascript" src="src/Song.js"></script>
```

Note: Below this section of SpecRunner is code responsible for running the actual tests. Given that we won't be covering modifying this code, I'm going to skip reviewing it. I do however encourage you to take a look through PlayerSpec.js and SpecHelper.js. They're a useful basic example to go through how a minimal set of tests might work.

TDD With Backbone

When developing applications with Backbone, it can be necessary to test both individual modules of code as well as modules, views, collections and routers. Taking a TDD approach to testing, let's review some specs for testing these Backbone components using the popular Backbone Todo application. For this section we will be using a modified version of Larry Myers Backbone Koans project, which can be found in the `practicals\jasmine-koans` folder.

Models

The complexity of Backbone models can vary greatly depending on what your application is trying to achieve. In the following example, we're going to test default values, attributes, state changes and validation rules.

First, we begin our suite for model testing using `describe()`:

```
describe('Tests for Todo', function() {
```

Models should ideally have default values for attributes. This helps ensure that when creating instances without a value set for any specific attribute, a default one (e.g ”) is used instead. The idea here is to allow your application to interact with models without any unexpected behavior.

In the following spec, we create a new Todo without any attributes passed then check to find out what the value of the `text` attribute is. As no value has been set, we expect a default value of `"` to be returned.

```
it('Can be created with default values for its attributes.', function() {
    var todo = new Todo();
    expect(todo.get('text')).toBe('');
});
```

If testing this spec before your models have been written, you'll incur a failing test, as expected. What's required for the spec to pass is a default value for the attribute `text`. We can implement this default value with some other useful defaults (which we'll be using shortly) in our Todo model as follows:

```
window.Todo = Backbone.Model.extend({

    defaults: {
      text: '',
      done:  false,
      order: 0
    }
```

Next, we want to test that our model will pass attributes that are set such that retrieving the value of these attributes after initialization will be what we expect. Notice that here, in addition to testing for an expected value for `text`, we're also testing the other default values are what we expect them to be.

```
it('Will set passed attributes on the model instance when created.', function() {
    var todo = new Todo({ text: 'Get oil change for car.' });

    // what are the values expected here for each of the
    // attributes in our Todo?

    expect(todo.get('text')).toBe('Get oil change for car.'');
    expect(todo.get('done')).toBe(false);
    expect(todo.get('order')).toBe(0);
});
```

Backbone models support a model.change() event which is triggered when the state of a model changes. In the following example, by 'state' I'm referring to the value of a Todo model's attributes. The reason changes of state are important to test are that there may be state-dependent events in your application e.g you may wish to display a confirmation view once a Todo model has been updated.

```
it('Fires a custom event when the state changes.', function() {

    var spy = jasmine.createSpy('-change event callback-');
    var todo = new Todo();

    // how do we monitor changes of state?
    todo.on('change', spy);

    // what would you need to do to force a change of state?
    todo.set({ text: 'Get oil change for car.' });

    expect(spy).toHaveBeenCalled();
});
```

It's common to include validation logic in your models to ensure both the input passed from users (and other modules) in the application are 'valid'. A Todo app may wish to validate the text input supplied in case it contains rude words. Similarly if we're storing the `done` state of a Todo item using booleans, we need to validate that truthy/falsy values are passed and not just any arbitrary string.

In the following spec, we take advantage of the fact that validations which fail model.validate() trigger an "error" event. This allows us to test if validations are correctly failing when invalid input is supplied.

We create an errorCallback spy using Jasmine's built in `createSpy()` method which allows us to spy on the error event as follows:

```
it('Can contain custom validation rules, and will trigger an error event on failed validatio

    var errorCallback = jasmine.createSpy('-error event callback-');

    var todo = new Todo();

    todo.on('error', errorCallback);

    // What would you need to set on the todo properties to
    // cause validation to fail?

    todo.set({done:'a non-integer value'});

    var errorArgs = errorCallback.mostRecentCall.args;

    expect(errorArgs).toBeDefined();
    expect(errorArgs[0]).toBe(todo);
    expect(errorArgs[1]).toBe('Todo.done must be a boolean value.');
});
```

The code to make the above failing test support validation is relatively simple. In our model, we override the validate() method (as recommended in the Backbone docs), checking to make sure a model both has a 'done' property and is a valid boolean before allowing it to pass.

```
validate: function(attrs) {
    if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
        return 'Todo.done must be a boolean value.';
    }
}
```

If you would like to review the final code for our Todo model, you can find it below:

```
var NAUGHTY_WORDS = /crap|poop|hell|frogs/gi;

function sanitize(str) {
    return str.replace(NAUGHTY_WORDS, 'rainbows');
}

window.Todo = Backbone.Model.extend({

    defaults: {
      text: '',
      done:  false,
      order: 0
    },

    initialize: function() {
        this.set({text: sanitize(this.get('text'))}, {silent: true});
    },

    validate: function(attrs) {
        if (attrs.hasOwnProperty('done') && !_.isBoolean(attrs.done)) {
            return 'Todo.done must be a boolean value.';
        }
    },

    toggle: function() {
        this.save({done: !this.get('done')});
    }

});
```

Collections

We now need to define specs to tests a Backbone collection of Todo models (a TodoList). Collections are responsible for a number of list tasks including managing order and filtering.

A few specific specs that come to mind when working with collections are:

- Making sure we can add new Todo models as both objects and arrays
- Attribute testing to make sure attributes such as the base URL of the collection are values we expect
- Purposefully adding items with a status of `done:true` and checking against how many items the collection thinks have been completed vs. those that are remaining

In this section we're going to cover the first two of these with the third left as an extended exercise I recommend trying out.

Testing Todo models can be added to a collection as objects or arrays is relatively trivial. First, we initialize a new TodoList collection and check to make sure its length (i.e the number of Todo models it contains) is 0. Next, we add new Todos, both as objects and arrays, checking the length property of the collection at each stage to ensure the overall count is what we expect:

```
describe('Tests for TodoList', function() {

    it('Can add Model instances as objects and arrays.', function() {
        var todos = new TodoList();

        expect(todos.length).toBe(0);

        todos.add({ text: 'Clean the kitchen' });

        // how many todos have been added so far?
        expect(todos.length).toBe(1);

        todos.add([
            { text: 'Do the laundry', done: true },
            { text: 'Go to the gym'}
        ]);

        // how many are there in total now?
        expect(todos.length).toBe(3);
    });
...
```

Similar to model attributes, it's also quite straight-forward to test attributes in collections. Here we have a spec that ensures the collection.url (i.e the url reference to the collection's location on the server) is what we expect it to be:

```
it('Can have a url property to define the basic url structure for all contained models.', fu
        var todos = new TodoList();

        // what has been specified as the url base in our model?
        expect(todos.url).toBe('/todos/');
});
```

For the third spec, it's useful to remember that the implementation for our collection will have methods for filtering how many Todo items are done and how many are remaining - we can call these `done()` and `remaining()`. Consider writing a spec which creates a new collection and adds one new model that has a preset `done` state of `true` and two others that have the default `done` state of `false`. Testing the length of what's returned using `done()` and `remaining()` should allow us to know whether the state management in our application is working or needs a little tweaking.

The final implementation for our TodoList collection can be found below:

```
 window.TodoList = Backbone.Collection.extend({

        model: Todo,

        url: '/todos/',

        done: function() {
            return this.filter(function(todo) { return todo.get('done'); });
        },

        remaining: function() {
            return this.without.apply(this, this.done());
        },

        nextOrder: function() {
            if (!this.length) {
                return 1;
            }

            return this.last().get('order') + 1;
        },

        comparator: function(todo) {
            return todo.get('order');
        }

    });
```

Views

Before we take a look at testing Backbone views, let's briefly review a jQuery plugin that can assist with writing Jasmine specs for them.

The Jasmine jQuery Plugin

As we know our Todo application will be using jQuery for DOM manipulation, there's a useful jQuery plugin called jasmine-jquery we can use to help simplify BDD testing rendered elements that our views may produce.

The plugin provides a number of additional Jasmine matchers to help test jQuery wrapped sets such as:

- `toBe(jQuerySelector)` e.g `expect($('<div id="some-id"></div>')).toBe('div#some-id')`
- `toBeChecked()` e.g `expect($('<input type="checkbox" checked="checked"/>')).toBeChecked()`
- `toBeSelected()` e.g `expect($('<option selected="selected"></option>')).toBeSelected()`

and many others. The complete list of matchers supported can be found on the project homepage. It's useful to know that similar to the standard Jasmine matchers, the custom matchers above can be inverted using the .not prefix (i.e `expect(x).not.toBe(y)`):

```
expect($('<div>I am an example</div>')).not.toHaveText(/other/)
```

jasmine-jquery also includes a fixtures model, allowing us to load in arbitrary HTML content we may wish to use in our tests. Fixtures can be used as follows:

Include some HTML in an external fixtures file:

some.fixture.html: `<div id="sample-fixture">some HTML content</div>`

Next, inside our actual test we would load it as follows:

```
loadFixtures('some.fixture.html')
$('some-fixture').myTestedPlugin();
expect($('#some-fixture')).to<the rest of your matcher would go here>
```

The jasmine-jquery plugin is by default setup to load fixtures from a specific directory: spec/javascripts/fixtures. If you wish to configure this path you can do so by initially setting `jasmine.getFixtures().fixturesPath = 'your custom path'`.

Finally, jasmine-jquery includes support for spying on jQuery events without the need for any extra plumbing work. This can be done using the `spyOnEvent()` and `assert(eventName).toHaveBeenTriggered(selector)` functions. An example of usage may look as follows:

```
spyOnEvent($('#el'), 'click');
$('#el').click();
expect('click').toHaveBeenTriggeredOn($('#el'));
```

View testing

In this section we will review three dimensions to writing specs for Backbone Views: initial setup, view rendering and finally templating. The latter two of these are the most commonly tested, however we'll review shortly why writing specs for the initialization of your views can also be of benefit.

Initial setup

At their most basic, specs for Backbone views should validate that they are being correctly tied to specific DOM elements and are backed by valid data models. The reason to consider doing this is that failures to such specs can trip up more complex tests later on and they're fairly simple to write, given the overall value offered.

To help ensure a consistent testing setup for our specs, we use `beforeEach()` to append both an empty `UL` (#todoList) to the DOM and initialize a new instance of a TodoView using an empty Todo model. `afterEach()` is used to remove the previous #todoList `UL` as well as the previous instance of the view.

```
describe('Tests for TodoView', function() {

    beforeEach(function() {
        $('body').append('<ul id="todoList"></ul>');
        this.todoView = new TodoView({ model: new Todo() });
    });


    afterEach(function() {
        this.todoView.remove();
        $('#todoList').remove();
    });

...
```

The first spec useful to write is a check that the TodoView we've created is using the correct `tagName` (element or className). The purpose of this test is to make sure it's been correctly tied to a DOM element when it was created.

Backbone views typically create empty DOM elements once initialized, however these elements are not attached to the visible DOM in order to allow them to be constructed without an impact on the performance of rendering.

```
it('Should be tied to a DOM element when created, based off the property provided.', functio
    //what html element tag name represents this view?
    expect(todoView.el.tagName.toLowerCase()).toBe('li');
});
```

Once again, if the TodoView has not already been written, we will experience failing specs. Thankfully, solving this is as simple as creating a new Backbone.View with a specific `tagName`.

```
var todoView = Backbone.View.extend({
    tagName:  'li'
});
```

If instead of testing against the `tagName` you would prefer to use a className instead, we can take advantage of jasmine-jquery's `toHaveClass()` matcher to cater for this.

```
it('Should have a class of "todos"'), function(){
   expect(this.view.$el).toHaveClass('todos');
});
```

The `toHaveClass()` matcher operates on jQuery objects and if the plugin hadn't been used, an exception would have been incurred (it is of course also possible to test for the className by accessing el.className if not opting to use jasmine-jquery).

You may have noticed that in `beforeEach()`, we passed our view an initial (albeit unfilled) Todo model. Views should be backed by a model instance which provides data. As this is quite important to our view's ability to function, we can write a spec to ensure a model is both defined (using the `toBeDefined()` matcher) and then test attributes of the model to ensure defaults both exist and are the value we expect them to be.

```
it('Is backed by a model instance, which provides the data.', function() {

    expect(todoView.model).toBeDefined();

    // what's the value for Todo.get('done') here?
    expect(todoView.model.get('done')).toBe(false); //or toBeFalsy()
});
```

View rendering

Next we're going to take a look at writing specs for view rendering. Specifically, we want to test that our TodoView elements are actually rendering as expected.

In smaller applications, those new to BDD might argue that visual confirmation of view rendering could replace unit testing of views. The reality is that when dealing with applications that might grow to multiple-views, it often makes sense to automate this process as much as possible from the get-go. There are also aspects of rendering that require verification beyond what is visually presented on-screen (which we'll see very shortly).

We're going to begin testing views by writing two specs. The first spec will check that the view's `render()` method is correctly returning the view instance, which is necessary for chaining. Our second spec will check that the HTML produced is exactly what we expect based on the properties of the model instance that's been associated with our TodoView.

Unlike some of the previous specs we've covered, this section will make greater use of `beforeEach()` to both demonstrate how to use nested suites and also ensure a consistent set of conditions for our specs. In our first view spec for TodoView, we're simply going to create a sample model (based on Todo) and instantiate a TodoView which associates it with the model.

```
describe('TodoView', function() {

  beforeEach(function() {
    this.model = new Backbone.Model({
      text: 'My Todo',
      order: 1,
      done: false
    });
    this.view = new TodoView({model:this.model});
  });

  describe('Rendering', function() {

    it('returns the view object', function() {
      expect(this.view.render()).toEqual(this.view);
    });

    it('produces the correct HTML', function() {
      this.view.render();

      //let's use jasmine-jquery's toContain() to avoid
      //testing for the complete content of a todo's markup
      expect(this.view.el.innerHTML)
```

```
        .toContain('<label class="todo-content">My Todo</label>');
    });

  });

});
```

Once these specs are run, only the second one ('produces the correct HTML') fails. Our first spec ('returns the view object'), which is testing that the TodoView instance is returned from `render()`, only passed as this is Backbone's default behavior. We haven't yet overwritten the `render()` method with our own version.

Note: For the purposes of maintaining readability, all template examples in this section will use a minimal version of the following Todo view template. As it's relatively trivial to expand this, please feel free to refer to this sample if needed:

```
<div class="todo <%= done ? 'done' : '' %>">
        <div class="display">
          <input class="check" type="checkbox" <%= done ? 'checked="checked"' : '' %> />
          <label class="todo-content"><%= text %></label>
          <span class="todo-destroy"></span>
        </div>
        <div class="edit">
          <input class="todo-input" type="text" value="<%= content %>" />
        </div>
</div>
```

The second spec fails with the following message:

`Expected " to contain '<label class="todo-content">My Todo</label>'.`

The reason for this is the default behavior for render() doesn't create any markup. Let's write a replacement for render() which fixes this:

```
render: function() {
  var template = '<label class="todo-content"><%= text %></label>';
  var output = template
    .replace('<%= text %>', this.model.get('text'));
  this.$el.html(output);
  return this;
}
```

The above specifies an inline string template and replaces fields found in the template within the "<% %>" blocks with their corresponding values from the associated model. As we're now also returning the TodoView instance from the method, the first spec will also pass. It's worth noting that there are serious drawbacks to using HTML strings in your specs to test against like this. Even minor changes to your template (a simple tab or whitespace) would cause your spec to fail, despite the rendered output being the same. It's also more time consuming to maintain as most templates in real-world applications are significantly more complex. A better option for testing rendered output is using jQuery to both select and inspect values.

With this in mind, let's re-write the specs, this time using some of the custom matchers offered by jasmine-jquery:

```
describe('Template', function() {

  beforeEach(function() {
    this.view.render();
  });

  it('has the correct text content', function() {
    expect(this.view.$('.todo-content'))
      .toHaveText('My Todo');
  });

});
```

It would be impossible to discuss unit testing without mentioning fixtures. Fixtures typically contain test data (e.g HTML) that is loaded in when needed (either locally or from an external file) for unit testing. So far we've been establishing jQuery expectations based on the view's el property. This works for a number of cases, however, there are instances where it may be necessary to render markup into the document. The most optimal way to handle this within specs is through using fixtures (another feature brought to us by the jasmine-jquery plugin).

Re-writing the last spec to use fixtures would look as follows:

```
describe('TodoView', function() {

  beforeEach(function() {
    ...
    setFixtures('<ul class="todos"></ul>');
  });

  ...

  describe('Template', function() {
    beforeEach(function() {
      $('.todos').append(this.view.render().el);
    });

    it('has the correct text content', function() {
      expect($('.todos').find('.todo-content'))
        .toHaveText('My Todo');
    });

  });

});
```

What we're now doing in the above spec is appending the rendered todo item into the fixture. We then set expectations against the fixture, which may be something desirable when a view is setup against an element which already exists in the DOM. It would be necessary to provide both the fixture and test the `el` property correctly picking up the element expected when the view is instantiated.

Rendering with a templating system

JavaScript templating systems (such as Handlebars, Mustache and even Underscore's own Micro-templating) support conditional logic in template strings. What this effectively means is that we can add if/else/ternery expressions inline which can then be evaluated as needed, allowing us to build even more powerful templates.

In our case, when a user sets a Todo item to be complete (done), we may wish to provide them with visual feedback (such as a striked line through the text) to differentiate the item from those that are remaining. This can be done by attaching a new class to the item. Let's begin by writing a test we would ideally like to work:

```
describe('When a todo is done', function() {

  beforeEach(function() {
    this.model.set({done: true}, {silent: true});
    $('.todos').append(this.view.render().el);
  });

  it('has a done class', function() {
    expect($('.todos .todo-content:first-child'))
      .toHaveClass('done');
  });
});
```

This will fail with the following message:

```
Expected '<label class="todo-content">My Todo</label>' to have
class 'done'.
```

which can be fixed in the existing render() method as follows:

```
render: function() {
  var template = '<label class="todo-content">' +
    '<%= text %></label>';
  var output = template
    .replace('<%= text %>', this.model.get('text'));
  this.$el.html(output);
  if (this.model.get('done')) {
    this.$('.todo-content').addClass('done');
  }
  return this;
}
```

This can however get unwieldily fairly quickly. As the logic in our templates increases, so does the complexity involved. This is where templates libraries can help. As mentioned earlier, there are a number of popular options available, but for the purposes of this chapter we're going to stick to using Underscore's built-in Microtemplating. Whilst there are more advanced options you're free to explore, the benefit of this is that no additional files are required and we can easily change the existing Jasmine specs without too much adjustment.

The TodoView object modified to use Underscore templating would look as follows:

```
var TodoView = Backbone.View.extend({

  tagName: 'li',

  initialize: function(options) {
    this.template = _.template(options.template || '');
  },

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

});
```

Above, the initialize() method compiles a supplied Underscore template (using the _.template() function) in the instantiation. A more common way of referencing templates is placing them in a script tag using a custom script type (e.g type="text/template"). As this isn't a script type any browser understands, it's simply ignored, however referencing the script by an id attribute allows the template to be kept separate to other parts of the page which wish to use it. In real world applications, it's preferable to either do this or load in templates stored in external files for testing.

For testing purposes, we're going to continue using the string injection approach to keep things simple. There is however a useful trick that can be applied to automatically create or extend templates in the Jasmine scope for each test. By creating a new directory (say, 'templates') in the 'spec' folder and adding a new script file with the following contents, to jasmine.yml or SpecRunner.html, we can add a todo property which contains the Underscore template we wish to use:

```
beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
    todo: '<label class="todo-content">' +
            '<%= text %>' +
          '</label>'
  });
});
```

To finish this off, we simply update our existing spec to reference the template when instantiating the TodoView object:

```
describe('TodoView', function() {

  beforeEach(function() {
    ...
    this.view = new TodoView({
      model: this.model,
      template: this.templates.todo
    });
  });

  ...

});
```

The existing specs we've looked at would continue to pass using this approach, leaving us free to adjust the template with some additional conditional logic for Todos with a status of 'done':

```
beforeEach(function() {
  this.templates = _.extend(this.templates || {}, {
    todo: '<label class="todo-content <%= done ? 'done' : '' %>"' +
            '<%= text %>' +
          '</label>'
  });
});
```

This will now also pass without any issues. Remember that jasmine-jquery also supports loading external fixtures into your specs easily using its build in `loadFixtures()` and `readFixtures()` methods. For more information, consider reading the official jasmine-jquery docs.

Conclusions

We have now covered how to write Jasmine tests for models, views and collections with Backbone.js. Whilst testing routing can at times be desirable, some developers feel it can be more optimal to leave this to third-party tools such as Selenium, so do keep this in mind.

James Newbery was kind enough to help me with writing the Views section above and his articles on Testing Backbone Apps With SinonJS were of great inspiration (you'll actually find some Handlebars examples of the view specs in part 3 of his article). If you would like to learn more about writing spies and mocks for Backbone using SinonJS as well as how to test Backbone routers, do consider reading his series.

Exercise

As an exercise, I recommend now trying the Jasmine Koans in `practicals\jasmine-joans` and trying to fix some of the purposefully failing tests it has to offer. This is an excellent way of not just learning how Jasmine specs and suites work, but working through the examples (without peeking back) will also put your Backbone skills to test too.

Further reading

- Jasmine + Backbone Revisited
- Backbone, PhantomJS and Jasmine

Unit Testing Backbone Applications With QUnit And SinonJS

Introduction

QUnit is a powerful JavaScript test suite written by jQuery team member Jörn Zaefferer and used by many large open-source projects (such as jQuery and Backbone.js) to test their code. It's both capable of testing standard JavaScript code in the browser as well as code on the server-side (where environments supported include Rhino, V8 and SpiderMonkey). This makes it a robust solution for a large number of use-cases.

Quite a few Backbone.js contributors feel that QUnit is a better introductory framework for testing if you don't wish to start off with Jasmine and BDD right away. As we'll see later on in this chapter, QUnit can also be combined with third-party solutions such as SinonJS to produce an even more powerful testing solution supporting spies and mocks, which some say is preferable over Jasmine.

My personal recommendation is that it's worth comparing both frameworks and opting for the solution that you feel the most comfortable with.

QUnit

Getting Setup

Luckily, getting QUnit setup is a fairly straight-forward process that will take less than 5 minutes.

We first setup a testing environment composed of three files:

- A HTML **structure** for displaying test results,
- The **qunit.js** file composing the testing framework and,
- The **qunit.css** file for styling test results.

The latter two of these can be downloaded from the QUnit website.

If you would prefer, you can use a hosted version of the QUnit source files for testing purposes. The hosted URLs can be found at [http://github.com/jquery/qunit/raw/master/qunit/].

Sample HTML with QUnit-compatible markup:

```
<!DOCTYPE html>
<html>
<head>
```

```
    <title>QUnit Test Suite</title>

     <link rel="stylesheet" href="qunit.css">
     <script src="qunit.js"></script>

     <!-- Your application -->
     <script src="app.js"></script>

     <!-- Your tests -->
     <script src="tests.js"></script>
</head>
<body>
    <h1 id="qunit-header">QUnit Test Suite</h1>
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-toolbar"></div>
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests">test markup, hidden.</ol>
</body>
</html>
```

Let's go through the elements above with qunit mentioned in their ID. When QUnit is running:

- **qunit-header** shows the name of the test suite
- **qunit-banner** shows up as red if a test fails and green if all tests pass
- **qunit-testrunner-toolbar** contains additional options for configuring the display of tests
- **qunit-userAgent** displays the navigator.userAgent property
- **qunit-tests** is a container for our test results

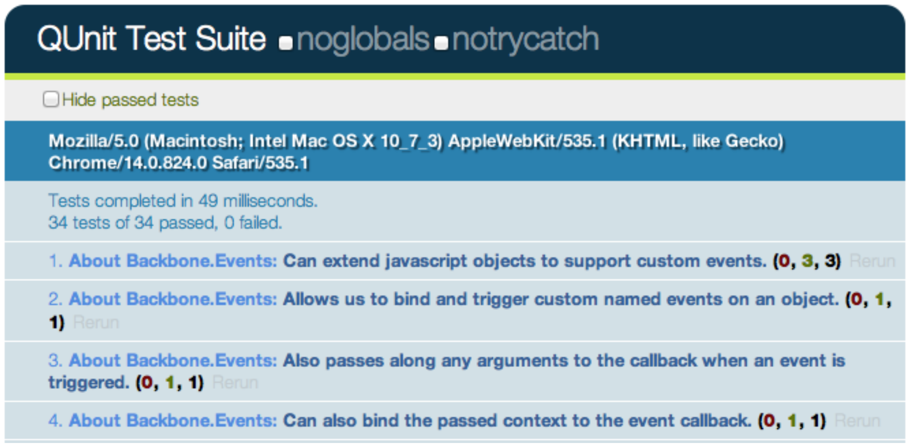
When running correctly, the above test runner looks as follows:

The numbers of the form (a, b, c) after each test name correspond to a) failed asserts, b) passed asserts and c) total asserts. Clicking on a test name expands it to display all of the assertions for that test case. Assertions in green have successfully passed.

If however any tests fail, the test gets highlighted (and the qunit-banner at the top switches to red):

Assertions

QUnit supports a number of basic **assertions**, which are used in testing to verify that the result being returned by our code is what we expect. If an assertion fails, we know that a bug exists.Similar to Jasmine, QUnit can be used to easily test



QUnit Test Suite ■noglobals ■notrycatch

☐Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.824.0 Safari/535.1

Tests completed in 49 milliseconds.
34 tests of 34 passed, 0 failed.

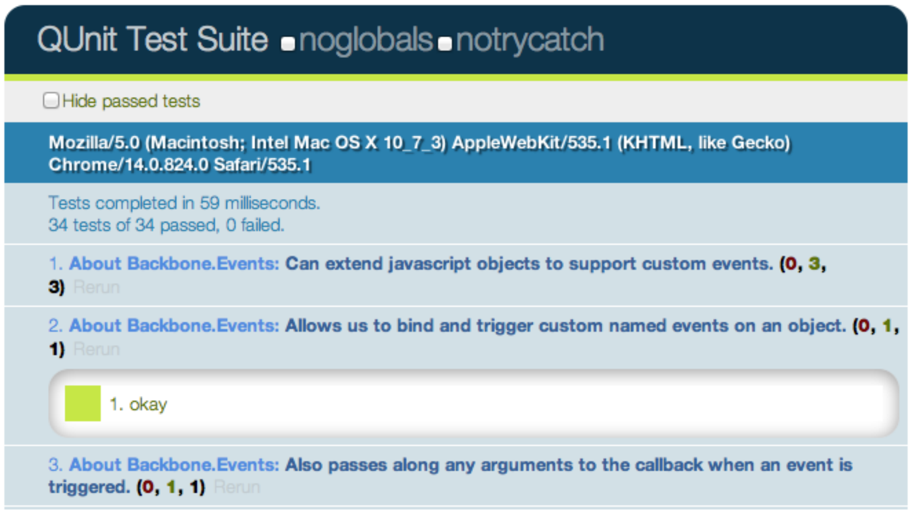
1. About Backbone.Events: Can extend javascript objects to support custom events. (0, 3, 3) Rerun

2. About Backbone.Events: Allows us to bind and trigger custom named events on an object. (0, 1, 1) Rerun

3. About Backbone.Events: Also passes along any arguments to the callback when an event is triggered. (0, 1, 1) Rerun

4. About Backbone.Events: Can also bind the passed context to the event callback. (0, 1, 1) Rerun

Figure 1: screenshot 1



QUnit Test Suite ■noglobals ■notrycatch

☐Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.824.0 Safari/535.1

Tests completed in 59 milliseconds.
34 tests of 34 passed, 0 failed.

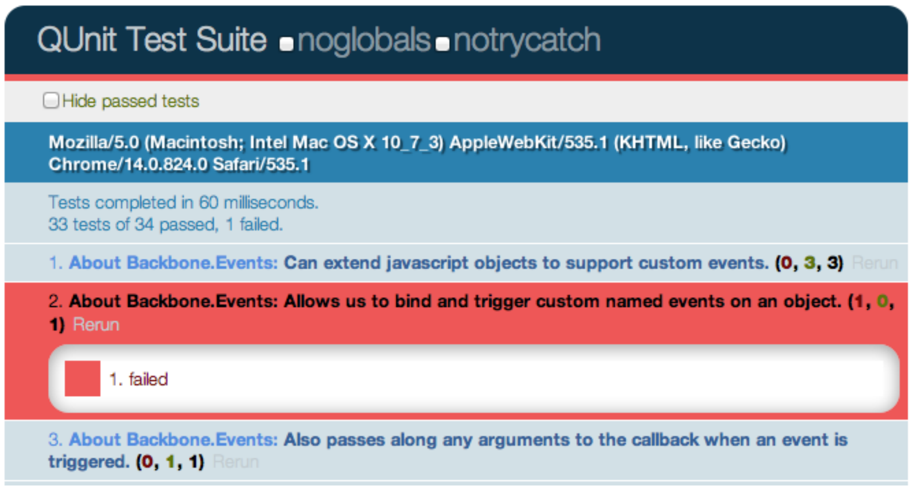
1. About Backbone.Events: Can extend javascript objects to support custom events. (0, 3, 3) Rerun

2. About Backbone.Events: Allows us to bind and trigger custom named events on an object. (0, 1, 1) Rerun

1. okay

3. About Backbone.Events: Also passes along any arguments to the callback when an event is triggered. (0, 1, 1) Rerun

Figure 2: screenshot 2


QUnit Test Suite noglobals notrycatch

Hide passed tests

Mozilla/5.0 (Macintosh; Intel Mac OS X 10_7_3) AppleWebKit/535.1 (KHTML, like Gecko) Chrome/14.0.824.0 Safari/535.1

Tests completed in 60 milliseconds.
33 tests of 34 passed, 1 failed.

1. About Backbone.Events: Can extend javascript objects to support custom events. (0, 3, 3) Rerun

2. About Backbone.Events: Allows us to bind and trigger custom named events on an object. (1, 0, 1) Rerun

1. failed

3. About Backbone.Events: Also passes along any arguments to the callback when an event is triggered. (0, 1, 1) Rerun

Figure 3: screenshot 3

for regressions. Specifically, when a bug is found one can write an assertion to test the existence of the bug, write a patch and then commit both. If subsequent changes to the code break the test you'll know what was responsible and be able to address it more easily.

Some of the supported QUnit assertions we're going to look at first are:

- `ok ( state, message )` - passes if the first argument is truthy
- `equal ( actual, expected, message )` - a simple comparison assertion with type coercion
- `notEqual ( actual, expected, message )` - the opposite of the above
- `expect( amount )` - the number of assertions expected to run within each test
- `strictEqual( actual, expected, message) -` offers a much stricter comparison than `equal()` and is considered the preferred method of checking equality as it avoids stumbling on subtle coercion bugs
- `deepEqual( actual, expected, message ) -` similar to `strictEqual`, comparing the contents (with `===`) of the given objects, arrays and primitives.

Creating new test cases with QUnit is relatively straight-forward and can be done using `test()`, which constructs a test where the first argument is the `name` of the test to be displayed in our results and the second is a `callback` function containing all of our assertions. This is called as soon as QUnit is running.

Basic test case using test(name, callback):

```
var myString = 'Hello Backbone.js';

test( 'Our first QUnit test - asserting results', function(){

    // ok( boolean, message )
    ok( true, 'the test succeeds');
    ok( false, 'the test fails');

    // equal( actualValue, expectedValue, message )
    equal( myString, 'Hello Backbone.js', 'The value expected is Hello Backbone.js!');
});
```

What we're doing in the above is defining a variable with a specific value and then testing to ensure the value was what we expected it to be. This was done using the comparison assertion, `equal()`, which expects its first argument to be a value being tested and the second argument to be the expected value. We also used `ok()`, which allows us to easily test against functions or variables that evaluate to booleans.

Note: Optionally in our test case, we could have passed an 'expected' value to `test()` defining the number of assertions we expect to run. This takes the form: `test( name, [expected], test );` or by manually settings the expectation at the top of the test function, like so: `expect( 1 )`. I recommend you to make it a habit and always define how many assertions you expect. More on this later.

As testing a simple static variable is fairly trivial, we can take this further to test actual functions. In the following example we test the output of a function that reverses a string to ensure that the output is correct using `equal()` and `notEqual()`:

Comparing the actual output of a function against the expected output:

```
function reverseString( str ){
    return str.split('').reverse().join('');
}

test( 'reverseString()', function() {
    expect( 5 );
    equal( reverseString('hello'), 'olleh', 'The value expected was olleh' );
    equal( reverseString('foobar'), 'raboof', 'The value expected was raboof' );
    equal( reverseString('world'), 'dlrow', 'The value expected was dlrow' );
    notEqual( reverseString('world'), 'dlroo', 'The value was expected to not be dlroo' );
    equal( reverseString('bubble'), 'double', 'The value expected was elbbub' );
})
```

Running these tests in the QUnit test runner (which you would see when your HTML test page was loaded) we would find that four of the assertions pass whilst the last one does not. The reason the test against `'double'` fails is because it was purposefully written incorrectly. In your own projects if a test fails to pass and your assertions are correct, you've probably just found a bug!

Adding structure to assertions

Housing all of our assertions in one test case can quickly become difficult to maintain, but luckily QUnit supports structuring blocks of assertions more cleanly. This can be done using `module()` - a method that allows us to easily group tests together. A typical approach to grouping might be keeping multiple tests testing a specific method as part of the same group (module).

Basic QUnit Modules:

```
module( 'Module One' );
test( 'first test', function() {} );
test( 'another test', function() {} );

module( 'Module Two' );
test( 'second test', function() {} );
test( 'another test', function() {} );

module( 'Module Three' );
test( 'third test', function() {} );
test( 'another test', function() {} );
```

We can take this further by introducing `setup()` and `teardown()` callbacks to our modules, where `setup()` is run before each test whilst `teardown()` is run after each test.

Using setup() and teardown() :

```
module( 'Module One', {
    setup: function() {
        // run before
    },
    teardown: function() {
        // run after
    }
});
```

```
test('first test', function() {
    // run the first test
});
```

These callbacks can be used to define (or clear) any components we wish to instantiate for use in one or more of our tests. As we'll see shortly, this is ideal for defining new instances of views, collections, models or routers from a project that we can then reference across multiple tests.

Using setup() and teardown() for instantiation and clean-up:

```
// Define a simple model and collection modeling a store and
// list of stores

var Store = Backbone.Model.extend({});

var StoreList = Backbone.Collection.extend({
    model: store,
    comparator: function( store ) { return store.get('name') }
});

// Define a group for our tests
module( 'StoreList sanity check', {
    setup: function() {
        this.list = new StoreList;
        this.list.add(new Store({ name: 'Costcutter' }));
        this.list.add(new Store({ name: 'Target' }));
        this.list.add(new Store({ name: 'Walmart' }));
        this.list.add(new Store({ name: 'Barnes & Noble' });
    },
    teardown: function() {
        window.errors = null;
    }
});

// Test the order of items added
test( 'test ordering', function() {
    expect( 1 );
    var expected = ['Barnes & Noble', 'Costcutter', 'Target', 'Walmart'];
    var actual = this.list.pluck('name');
    deepEqual( actual, expected, 'is maintained by comparator' );
});
```

Here, a list of stores is created and stored on `setup()`. A `teardown()` callback is used to simply clear our a list of errors we might be storing within the window scope, but is otherwise not needed.

Assertion examples

Before we continue any further, let's review some more examples of how QUnits various assertions can be correctly used when writing tests:

equal - a comparison assertion. It passes if actual == expected

```
test( 'equal', 2, function() {
  var actual = 6 - 5;
  equal( actual, true,  'passes as 1 == true' );
  equal( actual, 1,     'passes as 1 == 1' );
});
```

notEqual - a comparison assertion. It passes if actual != expected

```
test( 'notEqual', 2, function() {
  var actual = 6 - 5;
  notEqual( actual, false, 'passes as 1 != false' );
  notEqual( actual, 0,     'passes as 1 != 0' );
});
```

strictEqual - a comparison assertion. It passes if actual === expected.

```
test( 'strictEqual', 2, function() {
  var actual = 6 - 5;
  strictEqual( actual, true,  'fails as 1 !== true' );
  strictEqual( actual, 1,     'passes as 1 === 1' );
});
```

notStrictEqual - a comparison assertion. It passes if actual !== expected.

```
test('notStrictEqual', 2, function() {
  var actual = 6 - 5;
  notStrictEqual( actual, true,  'passes as 1 !== true' );
  notStrictEqual( actual, 1,     'fails as 1 === 1' );
});
```

deepEqual - a recursive comparison assertion. Unlike strictEqual(), it works on objects, arrays and primitives.

```
test('deepEqual', 4, function() {
  var actual = {q: 'foo', t: 'bar'};
  var el =  $('div');
  var children = $('div').children();

  equal( actual, {q: 'foo', t: 'bar'},   'fails - objects are not equal using equal()' );
  deepEqual( actual, {q: 'foo', t: 'bar'},   'passes - objects are equal' );
  equal( el, children, 'fails - jQuery objects are not the same' );
  deepEqual(el, children, 'fails - objects not equivalent' );

});
```

notDeepEqual - a comparison assertion. This returns the opposite of deepEqual

```
test('notDeepEqual', 2, function() {
  var actual = {q: 'foo', t: 'bar'};
  notEqual( actual, {q: 'foo', t: 'bar'},   'passes - objects are not equal' );
  notDeepEqual( actual, {q: 'foo', t: 'bar'},   'fails - objects are equivalent' );
});
```

raises - an assertion which tests if a callback throws any exceptions

```
test('raises', 1, function() {
  raises(function() {
    throw new Error( 'Oh no! It`s an error!' );
  }, 'passes - an error was thrown inside our callback');
});
```

Fixtures

From time to time we may need to write tests that modify the DOM. Managing the clean-up of such operations between tests can be a genuine pain, but thankfully QUnit has a solution to this problem in the form of the `#qunit-fixture` element, seen below.

Fixture markup:

```
<!DOCTYPE html>
<html>
```

```
<head>
    <title>QUnit Test</title>
    <link rel="stylesheet" href="qunit.css">
    <script src="qunit.js"></script>
    <script src="app.js"></script>
    <script src="tests.js"></script>
</head>
<body>
    <h1 id="qunit-header">QUnit Test</h1>
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-toolbar"></div>
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests"></ol>
    <div id="qunit-fixture"></div>
</body>
</html>
```

We can either opt to place static markup in the fixture or just insert/append any DOM elements we may need to it. QUnit will automatically reset the `innerHTML` of the fixture after each test to its original value. In case you're using jQuery, it's useful to know that QUnit checks for its availability and will opt to use `$(el).html()` instead, which will cleanup any jQuery event handlers too.

Fixtures example:

Let us now go through a more complete example of using fixtures. One thing that most of us are used to doing in jQuery is working with lists - they're often used to define the markup for menus, grids and a number of other components. You may have used jQuery plugins before that manipulated a given list in a particular way and it can be useful to test that the final (manipulated) output of the plugin is what was expected.

For the purposes of our next example, we're going to use Ben Alman's `$.enumerate()` plugin, which can prepend each item in a list by its index, optionally allowing us to set what the first number in the list is. The code snippet for the plugin can be found below, followed by an example of the output is generates:

```
$.fn.enumerate = function( start ) {
      if ( typeof start !== 'undefined' ) {
        // Since `start` value was provided, enumerate and return
        // the initial jQuery object to allow chaining.

        return this.each(function(i){
          $(this).prepend( '<b>' + ( i + start ) + '</b> ' );
        });

      } else {
        // Since no `start` value was provided, function as a
        // getter, returing the appropriate value from the first
        // selected element.

        var val = this.eq( 0 ).children( 'b' ).eq( 0 ).text();
        return Number( val );
      }
    };

/*
    <ul>
      <li>1. hello</li>
      <li>2. world</li>
      <li>3. i</li>
      <li>4. am</li>
      <li>5. foo</li>
    </ul>
*/
```

Let's now write some specs for the plugin. First, we define the markup for a list containing some sample items inside our `qunit-fixture` element:

```
&lt;div id=&quot;qunit-fixture&quot;&gt;
    &lt;ul&gt;
      &lt;li&gt;hello&lt;/li&gt;
      &lt;li&gt;world&lt;/li&gt;
      &lt;li&gt;i&lt;/li&gt;
      &lt;li&gt;am&lt;/li&gt;
      &lt;li&gt;foo&lt;/li&gt;
    &lt;/ul&gt;
  &lt;/div&gt;
```

Next, we need to think about what should be tested. `$.enumerate()` supports a few different use cases, including:

- **No arguments passed** - i.e `$(el).enumerate()`
- **0 passed as an argument** - i.e `$(el).enumerate(0)`
- **1 passed as an argument** - i.e `$(el).enumerate(1)`

As the text value for each list item is of the form "n. item-text" and we only require this to test against the expected output, we can simply access the content using `$(el).eq(index).text()` (for more information on .eq() see here).

and finally, here are our test cases:

```
module('jQuery#enumerate');

test( 'No arguments passed', 5, function() {
  var items = $('#qunit-fixture li').enumerate();
  equal( items.eq(0).text(), '1. hello', 'first item should have index 1' );
  equal( items.eq(1).text(), '2. world', 'second item should have index 2' );
  equal( items.eq(2).text(), '3. i', 'third item should have index 3' );
  equal( items.eq(3).text(), '4. am', 'fourth item should have index 4' );
  equal( items.eq(4).text(), '5. foo', 'fifth item should have index 5' );
});

test( '0 passed as an argument', 5, function() {
  var items = $('#qunit-fixture li').enumerate( 0 );
  equal( items.eq(0).text(), '0. hello', 'first item should have index 0' );
  equal( items.eq(1).text(), '1. world', 'second item should have index 1' );
  equal( items.eq(2).text(), '2. i', 'third item should have index 2' );
  equal( items.eq(3).text(), '3. am', 'fourth item should have index 3' );
  equal( items.eq(4).text(), '4. foo', 'fifth item should have index 4' );
});

test( '1 passed as an argument', 3, function() {
  var items = $('#qunit-fixture li').enumerate( 1 );
  equal( items.eq(0).text(), '1. hello', 'first item should have index 1' );
  equal( items.eq(1).text(), '2. world', 'second item should have index 2' );
  equal( items.eq(2).text(), '3. i', 'third item should have index 3' );
  equal( items.eq(3).text(), '4. am', 'fourth item should have index 4' );
  equal( items.eq(4).text(), '5. foo', 'fifth item should have index 5' );
});
```

Asynchronous code

As with Jasmine, the effort required to run synchronous tests with QUnit is fairly straight-forward. That said, what about tests that require asynchronous callbacks (such as expensive processes, Ajax requests and so on)? When we're dealing with asynchronous code, rather than letting QUnit control when the next test runs, we can inform that we need it to stop running and wait until it's okay to continue once again.

Remember: running asynchronous code without any special considerations can cause incorrect assertions to appear in other tests, so we want to make sure we get it right.

Writing QUnit tests for asynchronous code is made possible using the `start()` and `stop()` methods, which programmatically set the start and stop points during such tests. Here's a simple example:

```
test('An async test', function(){
   stop();
   expect( 1 );
   $.ajax({
        url: '/test',
        dataType: 'json',
        success: function( data ){
            deepEqual(data, {
               topic: 'hello',
               message: 'hi there!''
            });
            start();
        }
    });
});
```

A jQuery `$.ajax()` request is used to connect to a test resource and assert that the data returned is correct. `deepEqual()` is used here as it allows us to compare different data types (e.g objects, arrays) and ensures that what is returned is exactly what we're expecting. We know that our Ajax request is asynchronous and so we first call `stop()`, run the code making the request and finally at the very end of our callback, inform QUnit that it is okay to continue running other tests.

Note: rather than including `stop()`, we can simply exclude it and substitute `test()` with `asyncTest()` if we prefer. This improves readability when dealing with a mixture of asynchronous and synchronous tests in your suite. Whilst this setup should work fine for many use-cases, there is no guarantee that the callback in our `$.ajax()` request will actually get called. To factor this into our tests, we can use `expect()` once again to define how many assertions we expect to see within our test. This is a healthy safety blanket as it ensures that if a test completes with an insufficient number of assertions, we know something went wrong and fix it.

SinonJS

Similar to the section on testing Backbone.js apps using the Jasmine BDD framework, we're nearly ready to take what we've learned and write a number of QUnit tests for our Todo application.

Before we start though, you may have noticed that QUnit doesn't support test spies. Test spies are functions which record arguments, exceptions and return values for any of their calls. They're typically used to test callbacks and how functions may be used in the application being tested. In testing frameworks, spies can usually be either anonymous functions or wrap functions which already exist.

What is SinonJS?

In order for us to substitute support for spies in QUnit, we will be taking advantage of a mocking framework called SinonJS by Christian Johansen. We will also be using the SinonJS-QUnit adapter which provides seamless integration with QUnit (meaning setup is minimal). Sinon.JS is completely test-framework agnostic and should be easy to use with any testing framework, so it's ideal for our needs.

The framework supports three features we'll be taking advantage of for unit testing our application:

- **Anonymous spies**
- **Spying on existing methods**
- **A rich inspection interface**

Using `this.spy()` without any arguments creates an anonymous spy. This is comparable to `jasmine.createSpy()` and we can observe basic usage of a SinonJS spy in the following example:

Basic Spies:

```
test('should call all subscribers for a message exactly once', function () {
    var message = getUniqueString();
    var spy1 = this.spy();

    PubSub.subscribe( message, spy );
    PubSub.publishSync( message, 'Hello World' );

    ok( spy1.calledOnce, 'the subscriber was called once' );
});
```

We can also use `this.spy()` to spy on existing functions (like jQuery's `$.ajax`) in the example below. When spying on a function which already exists, the function behaves normally but we get access to data about its calls which can be very useful for testing purposes.

Spying On Existing Functions:

```
test( 'should inspect the jQuery.getJSON usage of jQuery.ajax', function () {
    this.spy( jQuery, 'ajax' );

    jQuery.getJSON( '/todos/completed' );

    ok( jQuery.ajax.calledOnce );
    equals( jQuery.ajax.getCall(0).args[0].url, '/todos/completed' );
    equals( jQuery.ajax.getCall(0).args[0].dataType, 'json' );
});
```

SinonJS comes with a rich spy interface which allows us to test whether a spy was called with a specific argument, if it was called a specific number of times and test against the values of arguments. A complete list of features supported in the interface can be found on SinonJS.org, but let's take a look at some examples demonstrating some of the most commonly used ones:

Matching arguments: test a spy was called with a specific set of arguments:

```
test( 'Should call a subscriber with standard matching': function () {
    var spy = sinon.spy();

    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', { id: 45 } );

    assertTrue( spy.calledWith( { id: 45 } ) );
});
```

Stricter argument matching: test a spy was called at least once with specific arguments and no others:

```
test( 'Should call a subscriber with strict matching': function () {
    var spy = sinon.spy();

    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', 'many', 'arguments' );
    PubSub.publishSync( 'message', 12, 34 );

    // This passes
    assertTrue( spy.calledWith('many') );

    // This however, fails
    assertTrue( spy.calledWithExactly( 'many' ) );
});
```

Testing call order: testing if a spy was called before or after another spy:

```
test( 'Should call a subscriber and maintain call order': function () {
    var a = sinon.spy();
    var b = sinon.spy();

    PubSub.subscribe( 'message', a );
    PubSub.subscribe( 'event', b );

    PubSub.publishSync( 'message', { id: 45 } );
    PubSub.publishSync( 'event', [1, 2, 3] );

    assertTrue( a.calledBefore(b) );
    assertTrue( b.calledAfter(a) );
});
```

Match execution counts: test a spy was called a specific number of times:

```
test( 'Should call a subscriber and check call counts', function () {
    var message = getUniqueString();
    var spy = this.spy();

    PubSub.subscribe( message, spy );
    PubSub.publishSync( message, 'some payload' );


    // Passes if spy was called once and only once.
    ok( spy.calledOnce ); // calledTwice and calledThrice are also supported

    // The number of recorded calls.
    equal( spy.callCount, 1 );

    // Directly checking the arguments of the call
    equals( spy.getCall(0).args[0], message );
});
```

Stubs and mocks

SinonJS also supports two other powerful features which are useful to be aware of: stubs and mocks. Both stubs and mocks implement all of the features of the spy API, but have some added functionality.

Stubs

A stub allows us to replace any existing behaviour for a specific method with something else. They can be very useful for simulating exceptions and are most often used to write test cases when certain dependencies of your code-base may not yet be written.

Let us briefly re-explore our Backbone Todo application, which contained a Todo model and a TodoList collection. For the purpose of this walkthrough, we want to isolate our TodoList collection and fake the Todo model to test how adding new models might behave.

We can pretend that the models have yet to be written just to demonstrate how stubbing might be carried out. A shell collection just containing a reference to the model to be used might look like this:

```
var TodoList = Backbone.Collection.extend({
    model: Todo
});

// Let's assume our instance of this collection is
this.todoList;
```

Assuming our collection is instantiating new models itself, it's necessary for us to stub the models constructor function for the the test. This can be done by creating a simple stub as follows:

```
this.todoStub = sinon.stub( window, 'Todo' );
```

The above creates a stub of the Todo method on the window object. When stubbing a persistent object, it's necessary to restore it to its original state. This can be done in a `teardown()` as follows:

```
this.todoStub.restore();
```

After this, we need to alter what the constructor returns, which can be efficiently done using a plain `Backbone.Model` constructor. Whilst this isn't a Todo model, it does still provide us an actual Backbone model.

```
teardown: function() {
    this.model = new Backbone.Model({
      id: 2,
      title: 'Hello world'
    });
    this.todoStub.returns( this.model );
});
```

The expectation here might be that this snippet would ensure our TodoList collection always instantiates a stubbed Todo model, but because a reference to the model in the collection is already present, we need to reset the model property of our collection as follows:

```
this.todoList.model = Todo;
```

The result of this is that when our TodoList collection instantiates new Todo models, it will return our plain Backbone model instance as desired. This allows us to write a spec for testing the addition of new model literals as follows:

```
module( 'Should function when instantiated with model literals', {

  setup:function() {

    this.todoStub = sinon.stub(window, 'Todo');
    this.model = new Backbone.Model({
      id: 2,
      title: 'Hello world'
    });

    this.todoStub.returns(this.model);
    this.todos = new TodoList();

    // Let's reset the relationship to use a stub
    this.todos.model = Todo;
    this.todos.add({
      id: 2,
      title: 'Hello world'
    });
  },

  teardown: function() {
    this.todoStub.restore();
  }

});
```

```
test('should add a model', function() {
    equal( this.todos.length, 1 );
});

test('should find a model by id', function() {
    equal( this.todos.get(5).get('id'), 5 );
  });
});
```

Mocks

Mocks are effectively the same as stubs, however they mock a complete API out and have some built-in expectations for how they should be used. The difference between a mock and a spy is that as the expectations for their use are pre-defined, it will fail if any of these are not met.

Here's a snippet with sample usage of a mock based on PubSubJS. Here, we have a `clearTodo()` method as a callback and use mocks to verify its behavior.

```
test('should call all subscribers when exceptions', function () {
    var myAPI = { clearTodo: function () {} };

    var spy = this.spy();
    var mock = this.mock( myAPI );
    mock.expects( 'clearTodo' ).once().throws();

    PubSub.subscribe( 'message', myAPI.clearTodo );
    PubSub.subscribe( 'message', spy );
    PubSub.publishSync( 'message', undefined );

    mock.verify();
    ok( spy.calledOnce );
});
```

Exercise

We can now begin writing test specs for our Todo application, which are listed and separated by component (e.g Models, Collections etc.). It's useful to pay attention to the name of the test, the logic being tested and most importantly the assertions being made as this will give you some insight into how what we've learned can be applied to a complete application.

To get the most out of this section, I recommend looking at the QUnit Koans included in the `practicals\qunit-koans` folder - this is a port of the Backbone.js Jasmine Koans over to QUnit that I converted for this post.

In case you haven't had a chance to try out one of the Koans kits as yet, they are a set of unit tests using a specific testing framework that both demonstrate how a set of specs for an application may be written, but also leave some tests unfilled so that you can complete them as an exercise.

Models

For our models we want to at minimum test that:

- New instances can be created with the expected default values
- Attributes can be set and retrieved correctly
- Changes to state correctly fire off custom events where needed
- Validation rules are correctly enforced

```
module( 'About Backbone.Model');

test('Can be created with default values for its attributes.', function() {
    expect( 1 );

    var todo = new Todo();

    equal( todo.get('text'), '' );
});

test('Will set attributes on the model instance when created.', function() {
    expect( 3 );

    var todo = new Todo( { text: 'Get oil change for car.' } );

    equal( todo.get('text'), 'Get oil change for car.' );
    equal( todo.get('done'), false );
    equal( todo.get('order'), 0 );
});

test('Will call a custom initialize function on the model instance when created.', function
    expect( 1 );

    var toot = new Todo({ text: 'Stop monkeys from throwing their own crap!' });
    equal( toot.get('text'), 'Stop monkeys from throwing their own rainbows!' );
});

test('Fires a custom event when the state changes.', function() {
    expect( 1 );
    var spy = this.spy();
    var todo = new Todo();

    todo.on( 'change', spy );
    // How would you update a property on the todo here?
    // Hint: http://documentcloud.github.com/backbone/#Model-set
    todo.set( { text: 'new text' } );

    ok( spy.calledOnce, 'A change event callback was correctly triggered' );
});


test('Can contain custom validation rules, and will trigger an error event on failed validat
    expect( 3 );

    var errorCallback = this.spy();
    var todo = new Todo();

    todo.on('error', errorCallback);
    // What would you need to set on the todo properties to cause validation to fail?
    todo.set( { done: 'not a boolean' } );

    ok( errorCallback.called, 'A failed validation correctly triggered an error' );
    notEqual( errorCallback.getCall(0), undefined );
    equal( errorCallback.getCall(0).args[1], 'Todo.done must be a boolean value.' );

});
```

Collections

For our collection we'll want to test that:

- New model instances can be added as both objects and arrays
- Changes to models result in any necessary custom events being fired
- A `url` property for defining the URL structure for models is correctly defined

```
module( 'About Backbone.Collection');

test( 'Can add Model instances as objects and arrays.', function() {
    expect( 3 );

    var todos = new TodoList();
    equal( todos.length, 0 );
    todos.add( { text: 'Clean the kitchen' } );
    equal( todos.length, 1 );

    todos.add([
        { text: 'Do the laundry', done: true },
        { text: 'Go to the gym' }
    ]);

    equal( todos.length, 3 );
});

test( 'Can have a url property to define the basic url structure for all contained models.'
    expect( 1 );
    var todos = new TodoList();
    equal( todos.url, '/todos/' );
});

test('Fires custom named events when the models change.', function() {
    expect(2);

    var todos = new TodoList();
    var addModelCallback = this.spy();
    var removeModelCallback = this.spy();

    todos.on( 'add', addModelCallback );
    todos.on( 'remove', removeModelCallback );

    // How would you get the 'add' event to trigger?
    todos.add( {text:'New todo'} );

    ok( addModelCallback.called );

    // How would you get the 'remove' callback to trigger?
    todos.remove( todos.last() );

    ok( removeModelCallback.called );
});
```

Views

For our views we want to ensure:

- They are being correctly tied to a DOM element when created

- They can render, after which the DOM representation of the view should be visible
- They support wiring up view methods to DOM elements

One could also take this further and test that user interactions with the view correctly result in any models that need to be changed being updated correctly.

```
module( 'About Backbone.View', {
    setup: function() {
        $('body').append('<ul id="todoList"></ul>');
        this.todoView = new TodoView({ model: new Todo() });
    },
    teardown: function() {
        this.todoView.remove();
        $('#todoList').remove();
    }
});

test('Should be tied to a DOM element when created, based off the property provided.', funct
    expect( 1 );
    equal( this.todoView.el.tagName.toLowerCase(), 'li' );
});

test('Is backed by a model instance, which provides the data.', function() {
    expect( 2 );
    notEqual( this.todoView.model, undefined );
    equal( this.todoView.model.get('done'), false );
});

test('Can render, after which the DOM representation of the view will be visible.', functio
   this.todoView.render();

    // Hint: render() just builds the DOM representation of the view, but doesn't insert it
    //       How would you append it to the ul#todoList?
    //       How do you access the view's DOM representation?
    //
    // Hint: http://documentcloud.github.com/backbone/#View-el

    $('ul#todoList').append(this.todoView.el);
    equal($('#todoList').find('li').length, 1);
});

asyncTest('Can wire up view methods to DOM elements.', function() {
    expect( 2 );
    var viewElt;
    $('#todoList').append( this.todoView.render().el );

    setTimeout(function() {
        viewElt = $('#todoList li input.check').filter(':first');

        equal(viewElt.length > 0, true);

        // Make sure that QUnit knows we can continue
        start();
    }, 1000, 'Expected DOM Elt to exist');


    // Hint: How would you trigger the view, via a DOM Event, to toggle the 'done' status.
    //       (See todos.js line 70, where the events hash is defined.)
    //
    // Hint: http://api.jquery.com/click

    $('#todoList li input.check').click();
    expect( this.todoView.model.get('done'), true );
});
```

Event

For events, we may want to test a few different use cases:

- Extending plain objects to support custom events
- Binding and triggering custom events on objects
- Passing along arguments to callbacks when events are triggered
- Binding a passed context to an event callback
- Removing custom events

and a few others that will be detailed in our module below:

```
module( 'About Backbone.Events', {
    setup: function() {
        this.obj = {};
        _.extend( this.obj, Backbone.Events );
        this.obj.off(); // remove all custom events before each spec is run.
    }
});

test('Can extend JavaScript objects to support custom events.', function() {
    expect(3);
    var basicObject = {};

    // How would you give basicObject these functions?
    // Hint: http://documentcloud.github.com/backbone/#Events
    _.extend( basicObject, Backbone.Events );

    equal( typeof basicObject.on, 'function' );
    equal( typeof basicObject.off, 'function' );
    equal( typeof basicObject.trigger, 'function' );
});

test('Allows us to bind and trigger custom named events on an object.', function() {
    expect( 1 );

    var callback = this.spy();

    this.obj.on( 'basic event', callback );
    this.obj.trigger( 'basic event' );

    // How would you cause the callback for this custom event to be called?
    ok( callback.called );
});

test('Also passes along any arguments to the callback when an event is triggered.', functio
    expect( 1 );

    var passedArgs = [];

    this.obj.on('some event', function() {
        for (var i = 0; i < arguments.length; i++) {
            passedArgs.push( arguments[i] );
        }
    });

    this.obj.trigger( 'some event', 'arg1', 'arg2' );

    deepEqual( passedArgs, ['arg1', 'arg2'] );
});


test('Can also bind the passed context to the event callback.', function() {
    expect( 1 );

    var foo = { color: 'blue' };
    var changeColor = function() {
        this.color = 'red';
    };

    // How would you get 'this.color' to refer to 'foo' in the changeColor function?
    this.obj.on( 'an event', changeColor, foo );
    this.obj.trigger( 'an event' );

    equal( foo.color, 'red' );
});

test( 'Uses `all` as a special event name to capture all events bound to the object.', funct
    expect( 2 );

    var callback = this.spy();

    this.obj.on( 'all', callback );
    this.obj.trigger( 'custom event 1' );
    this.obj.trigger( 'custom event 2' );

    equal( callback.callCount, 2 );
    equal( callback.getCall(0).args[0], 'custom event 1' );
});

test('Also can remove custom events from objects.', function() {
    expect( 5 );

    var spy1 = this.spy();
    var spy2 = this.spy();
    var spy3 = this.spy();

    this.obj.on( 'foo', spy1 );
    this.obj.on( 'bar', spy1 );
    this.obj.on( 'foo', spy2 );
    this.obj.on( 'foo', spy3 );

    // How do you unbind just a single callback for the event?
    this.obj.off( 'foo', spy1 );
    this.obj.trigger( 'foo' );

    ok( spy2.called );

    // How do you unbind all callbacks tied to the event with a single method
    this.obj.off( 'foo' );
    this.obj.trigger( 'foo' );

    ok( spy2.callCount, 1 );
    ok( spy2.calledOnce, 'Spy 2 called once' );
    ok( spy3.calledOnce, 'Spy 3 called once' );

    // How do you unbind all callbacks and events tied to the object with a single method?
    this.obj.off( 'bar' );
    this.obj.trigger( 'bar' );

    equal( spy1.callCount, 0 );
});
```

App

It can also be useful to write specs for any application bootstrap you may have in place. For the following module, our setup initiates and appends a TodoApp view and we can test anything from local instances of views being correctly defined to application interactions correctly resulting in changes to instances of local collections.

```
module( 'About Backbone Applications' , {
    setup: function() {
        Backbone.localStorageDB = new Store('testTodos');
        $('#qunit-fixture').append('<div id="app"></div>');
        this.App = new TodoApp({ appendTo: $('# <app') });
    },

    teardown: function() {
        this.App.todos.reset();
        $('# <app').remove();
    }
});

test('Should bootstrap the application by initializing the Collection.', function() {
    expect( 2 );

    notEqual( this.App.todos, undefined );
    equal( this.App.todos.length, 0 );
});

test( 'Should bind Collection events to View creation.' , function() {
      $('#new-todo').val( 'Foo' );
      $('#new-todo').trigger(new $.Event( 'keypress', { keyCode: 13 } ));

      equal( this.App.todos.length, 1 );
 });
```

Further Reading & Resources

That's it for this section on testing applications with QUnit and SinonJS. I encourage you to try out the QUnit Backbone.js Koans and see if you can extend some of the examples. For further reading consider looking at some of the additional resources below:

- **Test-driven JavaScript Development (book)**
- **SinonJS/QUnit Adapter**
- **SinonJS and QUnit**
- **Automating JavaScript Testing With QUnit**
- **Ben Alman's Unit Testing With QUnit**
- **Another QUnit/Backbone.js demo project**
- **SinonJS helpers for Backbone**

Resources

Books & Courses

- PeepCode: Backbone.js Basics
- CodeSchool: Anatomy Of Backbone
- Recipies With Backbone
- Backbone Patterns
- Backbone On Rails
- MVC In JavaScript With Backbone
- Backbone Tutorials
- Derick Baileys Resources For Learning Backbone

Extensions/Libraries

- MarionetteJS
- AuraJS
- Thorax
- Lumbar
- Backbone Layout Manager
- Backbone Boilerplate
- Backbone.ModelBinder
- Backbone Relational - for model relationships
- Backbone CouchDB
- Backbone Validations - HTML5 inspired validations

Conclusions

This concludes our voyage into the wondrous world of Backbone.js, but hopefully marks the beginning of your next journey as a user. What you have hopefully learned is that, beyond an accessible API there actually isn't a great deal to the library. Much of its elegance lies in its simplicity and that is why many developers use it.

For the simplest of applications, you are unlikely to need more than what is prescribed out of the box. For those developing complex applications however, the Backbone.js classes are straightforward to extend, providing an easy path to building (or using a pre-made) extension layer on top of it. We've experienced this first hand in the chapters on MarionetteJS and Thorax.

Working on the client-side can sometimes feel like the wild west, but I hope this book has introduced you to sound advice and concepts that will help you keep your code both tamed and a little more maintainable. Until next time, the very best of luck creating your own front-end masterpieces.

Notes

I would like to thank the Backbone.js, Stack Overflow, DailyJS (Alex Young) and JavaScript communities for their help, references and contributions to this book. This project would not be possible without you so thank you! :)

Appendix

MVP

Model-View-Presenter (MVP) is a derivative of the MVC design pattern which focuses on improving presentation logic. It originated at a company named Taligent in the early 1990s while they were working on a model for a C++ CommonPoint environment. Whilst both MVC and MVP target the separation of concerns across multiple components, there are some fundamental differences between them.

For the purposes of this summary we will focus on the version of MVP most suitable for web-based architectures.

Models, Views & Presenters

The P in MVP stands for presenter. It's a component which contains the user-interface business logic for the view. Unlike MVC, invocations from the view are delegated to the presenter, which are decoupled from the view and instead talk to it through an interface. This allows for all kinds of useful things such as being able to mock views in unit tests.

The most common implementation of MVP is one which uses a Passive View (a view which is for all intents and purposes "dumb"), containing little to no logic. MVP models are almost identical to MVC models and handle application data. The presenter acts as a mediator which talks to both the view and model, however both of these are isolated from each other. They effectively bind models to views, a responsibility held by Controllers in MVC. Presenters are at the heart of the MVP pattern and as you can guess, incorporate the presentation logic behind views.

Solicited by a view, presenters perform any work to do with user requests and pass data back to them. In this respect, they retrieve data, manipulate it and determine how the data should be displayed in the view. In some implementations, the presenter also interacts with a service layer to persist data (models). Models may trigger events but it's the presenter's role to subscribe to them so that it can update the view. In this passive architecture, we have no concept of direct data binding. Views expose setters which presenters can use to set data.

The benefit of this change from MVC is that it increases the testability of your application and provides a more clean separation between the view and the model. This isn't however without its costs as the lack of data binding support in the pattern can often mean having to take care of this task separately.

Although a common implementation of a Passive View is for the view to implement an interface, there are variations on it, including the use of events which can decouple the View from the Presenter a little more. As we don't have the interface construct in JavaScript, we're using it more and more as a protocol than an explicit interface here. It's technically still an API and it's probably fair for us to refer to it as an interface from that perspective.

There is also a Supervising Controller variation of MVP, which is closer to the MVC and MVVM - Model-View-ViewModel patterns as it provides data-binding from the Model directly from the View. Key-value observing (KVO) plugins (such as Derick Bailey's Backbone.ModelBinding plugin) introduce this idea of a Supervising Controller to Backbone.

MVP or MVC?

MVP is generally used most often in enterprise-level applications where it's necessary to reuse as much presentation logic as possible. Applications with very complex views and a great deal of user interaction may find that MVC doesn't quite fit the bill here as solving this problem may mean heavily relying on multiple controllers. In MVP, all of this complex logic can be encapsulated in a presenter, which can simplify maintenance greatly.

As MVP views are defined through an interface and the interface is technically the only point of contact between the system and the view (other than a presenter), this pattern also allows developers to write presentation logic without needing to wait for designers to produce layouts and graphics for the application.

Depending on the implementation, MVP may be more easy to automatically unit test than MVC. The reason often cited for this is that the presenter can be used as a complete mock of the user-interface and so it can be unit tested independent of other components. In my experience this really depends on the languages you are implementing MVP in (there's quite a difference between opting for MVP for a JavaScript project over one for say, ASP.NET).

At the end of the day, the underlying concerns you may have with MVC will likely hold true for MVP given that the differences between them are mainly semantic. As long as you are cleanly separating concerns into models, views and controllers (or presenters) you should be achieving most of the same benefits regardless of the pattern you opt for.

MVC, MVP and Backbone.js

There are very few, if any architectural JavaScript frameworks that claim to implement the MVC or MVP patterns in their classical form as many JavaScript developers don't view MVC and MVP as being mutually exclusive (we are actually more likely to see MVP strictly implemented when looking at web frameworks such as ASP.NET or GWT). This is because it's possible to have additional presenter/view logic in your application and yet still consider it a flavor of MVC.

Backbone contributor Irene Ros subscribes to this way of thinking as when she separates Backbone views out into their own distinct components, she needs something to actually assemble them for her. This could either be a controller route (such as a `Backbone.Router`, covered later in the book) or a callback in response to data being fetched.

That said, some developers do however feel that Backbone.js better fits the description of MVP than it does MVC . Their view is that:

- The presenter in MVP better describes the `Backbone.View` (the layer between View templates and the data bound to it) than a controller does
- The model fits `Backbone.Model` (it isn't that different from the classical MVC "Model")
- The views best represent templates (e.g Handlebars/Mustache markup templates)

A response to this could be that the view can also just be a View (as per MVC) because Backbone is flexible enough to let it be used for multiple purposes. The

V in MVC and the P in MVP can both be accomplished by `Backbone.View` because they're able to achieve two purposes: both rendering atomic components and assembling those components rendered by other views.

We've also seen that in Backbone the responsibility of a controller is shared with both the Backbone.View and Backbone.Router and in the following example we can actually see that aspects of that are certainly true.

Here, our Backbone `TodoView` uses the Observer pattern to 'subscribe' to changes to a View's model in the line `this.model.on('change',...)`. It also handles templating in the `render()` method, but unlike some other implementations, user interaction is also handled in the View (see `events`).

```
// The DOM element for a todo item...
app.TodoView = Backbone.View.extend({

  //... is a list tag.
  tagName:  'li',

  // Pass the contents of the todo template through a templating
  // function, cache it for a single todo
  template: _.template( $('#item-template').html() ),

  // The DOM events specific to an item.
  events: {
    'click .toggle':  'togglecompleted'
  },

  // The TodoView listens for changes to its model, re-rendering. Since there's
  // a one-to-one correspondence between a **Todo** and a **TodoView** in this
  // app, we set a direct reference on the model for convenience.
  initialize: function() {
    this.model.on( 'change', this.render, this );
    this.model.on( 'destroy', this.remove, this );
  },

  // Re-render the titles of the todo item.
  render: function() {
    this.$el.html( this.template( this.model.toJSON() ) );
    return this;
  },

  // Toggle the `"completed"` state of the model.
  togglecompleted: function() {
    this.model.toggle();
  },
});
```

Another (quite different) opinion is that Backbone more closely resembles Smalltalk-80 MVC, which we went through earlier.

As MarionetteJS author Derick Bailey has written, it's ultimately best not to force Backbone to fit any specific design patterns. Design patterns should be considered flexible guides to how applications may be structured and in this respect, Backbone doesn't fit either MVC nor MVP perfectly. Instead, it borrows some of the best concepts from multiple architectural patterns and creates a flexible framework that just works well. Call it **the Backbone way**, MV* or whatever helps reference its flavor of application architecture.

It *is* however worth understanding where and why these concepts originated, so I hope that my explanations of MVC and MVP have been of help. Most structural JavaScript frameworks will adopt their own take on classical patterns, either intentionally or by accident, but the important thing is that they help us develop applications which are organized, clean and can be easily maintained.

Upgrading to Backbone 0.9.10

Developing Backbone.js Applications is currently based on Backbone 0.9.2. If you are transitioning from this version to 0.9.10 or above, the following is a guide of changes grouped by classes, where applicable.

Note: We aim to update the entirety of this book to Backbone 1.0 once it has been tagged.

Model

- Model validation is now only enforced by default in `Model#save` and is no longer enforced by default upon construction or in `Model#set`, unless the `{validate:true}` option is passed:

```
var model = new Backbone.Model({name: "One"});
model.validate = function(attrs) {
  if (!attrs.name) {
    return "No thanks.";
  }
};
model.set({name: "Two"});
console.log(model.get('name'));
// 'Two'
model.unset('name', {validate: true});
// false
```

- Passing `{silent:true}` on change will no longer delay individual `"change:attr"` events, instead they are silenced entirely.

```
var model = new Backbone.Model();
model.set({x: true}, {silent: true});

console.log(!model.hasChanged(0));
// true
console.log(!model.hasChanged(''));
// true
```

- The `Model#change` method has been removed, as delayed attribute changes as no longer available.
- Calling `destroy` on a Model will now return `false` if the model `isNew`.

```
var model = new Backbone.Model();
console.log(model.destroy());
// false
```

- After fetching a model or a collection, all defined parse functions will now be run. So fetching a collection and getting back new models could cause both the collection to parse the list, and then each model to be parsed in turn, if you have both functions defined.
- HTTP PATCH support allows us to send only changed attributes (i.e partial updates) to the server by passing `{patch:  true}` i.e `model.save(attrs, {patch:  true})`.

```
// Save partial using PATCH
model.clear().set({id: 1, a: 1, b: 2, c: 3, d: 4});
model.save();
model.save({b: 2, d: 4}, {patch: true});
console.log(this.syncArgs.method);
// 'patch'
```

- When using `add` on a collection, passing `{merge:  true}` will now cause duplicate models to have their attributes merged in to the existing models, instead of being ignored.

```
var items = new Backbone.Collection;
items.add([{ id : 1, name: "Dog" , age: 3}, { id : 2, name: "cat" , age: 2}]);
items.add([{ id : 1, name: "Bear" }], {merge: true });
items.add([{ id : 2, name: "lion" }]); // merge: false

console.log(JSON.stringify(items.toJSON()));
// [{"id":1,"name":"Bear","age":3},{"id":2,"name":"cat","age":2}]
```

Collection

- While listening to a reset event, the list of previous models is now available in `options.previousModels`, for convenience.

```
var model = new Backbone.Model();
var collection = new Backbone.Collection([model])
.on('reset', function(collection, options) {
  console.log(options.previousModels);
  console.log([model]);
  console.log(options.previousModels[0] === model); // true
});
collection.reset([]);
```

- `Collection#sort` now triggers a `sort` event, instead of a `reset` event.
- Removed `getByCid` from Collections. `collection.get` now supports lookup by both id and cid.
- Collections now also proxy Underscore method name aliases (`collect`, `inject`, `foldl`, `foldr`, `head`, `tail`, `take`, and so on…)
- Added `update` (which is also available as an option to fetch) for "smart" updating of sets of models.

The update method attempts to perform smart updating of a collection using a specified list of models. When a model in this list isn't present in the collection, it is added. If it is, its attributes will be merged. Models which are present in the collection but not in the list are removed.

```
var theBeatles = new Collection(['john', 'paul', 'george', 'ringo']);

theBeatles.update(['john', 'paul', 'george', 'pete']);

// Fires a `remove` event for 'ringo', and an `add` event for 'pete'.
// Updates any of john, paul and georges's attributes that may have
// changed over the years.
```

- `collection.indexOf(model)` can be used to retrieve the index of a model as necessary.

```
var col = new Backbone.Collection;

col.comparator = function(a, b) {
  return a.get('name') < b.get('name') ? -1 : 1;
};

var tom = new Backbone.Model({name: 'Tom'});
var rob = new Backbone.Model({name: 'Rob'});
var tim = new Backbone.Model({name: 'Tim'});

col.add(tom);
col.add(rob);
col.add(tim);

console.log(col.indexOf(rob) === 0); // true
console.log(col.indexOf(tim) === 1); // true
console.log(col.indexOf(tom) === 2); // true
```

View

- `View#make` has been removed. You'll need to use `$` directly to construct DOM elements now.
- When declaring a View, `options`, `el`, `tagName`, `id` and `className` may now be defined as functions, if you want their values to be determined at runtime.

Events

- Backbone events now support jQuery-style event maps `obj.on({click: action})`. This is clearer than needing three separate calls to `.on` and should align better with the events hash used in Views:

```
 model.on({
     ‘change:name’ : this.nameChanged,
     ‘change:age’ : this.ageChanged,
     ‘change:height’ : this.heightChanges
 });
```

- The Backbone object now extends Events so that you can use it as a global event bus, if you like.
- Backbone events now supports once, similar to Node's once, or jQuery's one. A call to `once()` ensures that the callback only fires once when a notification arrives.

```
// Use once rather than having to explicitly unbind
var obj = { counterA: 0, counterB: 0 };
```

```
_.extend(obj, Backbone.Events);

var incrA = function(){ obj.counterA += 1; obj.trigger('event'); };
var incrB = function(){ obj.counterB += 1; };

obj.once('event', incrA);
obj.once('event', incrB);
obj.trigger('event');

console.log(obj.counterA === 1); // true
console.log(obj.counterB === 1); // true
```

`counterA` and `counterB` should only have been incremented once.

- Added listenTo and stopListening to Events. They can be used as inversion-of-control flavors of on and off, for convenient unbinding of all events an object is currently listening to. `view.remove()` automatically calls `view.stopListening()`.

If you've had a chance to work on a few Backbone projects by this point, you may know that every `on` called on an object also requires an `off` to be called in order for the garbage collector to do its job.

This can sometimes be overlooked when Views are binding to Models. In 0.9.10, this can now be done the other way around - Views can bind to Model notifications and unbind from all of them with just one call. We achieve this using `view.listenTo(model, 'eventName', func)` and `view.stopListening()`.

The default `remove()` of Views will call `stopListening()` for you, just in case you don't remember to.

```
var a = _.extend({}, Backbone.Events);
var b = _.extend({}, Backbone.Events);
a.listenTo(b, 'all', function(){ console.log(true); });
b.trigger('anything');
a.listenTo(b, 'all', function(){ console.log(false); });
a.stopListening();
b.trigger('anything');
```

A more complex example (from Just JSON) might require our Views to respond to "no connection" and "connection resume" events to re-fetch data on demand in an application.

In 0.9.2, we have to do this to achieve what we need:

```
 // In BaseView definition
var BaseView = Backbone.View.extend({
  destroy: function() {
    // Allow child views to hook to this event to unsubscribe
    // anything they may have subscribed to to other objects.
    this.trigger('beforedestroy');
    if (this.model) {
      this.model.off(null, null, this);
    }

    if (this.collection) {
      this.collection.off(null, null, this);
    }

    this.remove();
    this.unbind();
  }
});

// In MyView definition.
// We have a global EventBus that allows elements on the app to subscribe to global events.
// connection/disconnected, connection/resume is two of them.

var MyView = BaseView.extend({
  initialize: function() {
    this.on('beforedestroy', this.onBeforeDestroy, this);
    this.model.on('reset', this.onModelLoaded, this);
    EventBus.on('connection/disconnected', this.onDisconnect, this);
    EventBus.on('connection/resume', this.onConnectionResumed, this);
  },
  onModelLoaded: function() {
    // We only need this to be done once! (Kinda weird...)
    this.model.off('load', this.onModelLoaded, this);
  },
  onDisconnect: function() {
    // Figure out what state we are currently on, display View-specific messaging, etc.
  },
  onConnectionResumed: function() {
    // Re-do previous network request that failed.
  },
  onBeforeDestroy: function() {
    EventBus.off('connection/resume', this.onConnectionResumed, this);
    EventBus.off('connection/disconnected', this.onDisconnect, this);
  }
});
```

However, in 0.9.10, what we need to do is quite simple:

```
// In BaseView definition
var BaseView = Backbone.View.extend({
  destroy: function() {
    this.trigger('beforedestroy');
    this.remove();
  }
});

// In MyView definition.

var MyView = BaseView.extend({
  initialize: function() {
    this.listenTo(EventBus, 'connection/disconnected', this.onDisconnect);
    this.listenTo(EventBus, 'connection/resume', this.onConnectionResumed);
    this.once(this.model, 'load', this.onModelLoaded);
  },
  onModelLoaded: function() {
    // Don't need to unsubscribe anymore!
  },
  onDisconnect: function() {
    // Figure out the state, display messaging, etc.
  },
  onConnectionResumed: function() {
    // Re-do previous network request that failed.
  }
  // Most importantly, we no longer need onBeforeDestroy() anymore!
});
```

Routers

- A “route” event is triggered on the router in addition to being fired on Backbone.history.

```
Backbone.history.on('route', onRoute);

// Trigger 'route' event on router instance."
router.on('route', function(name, args) {
  console.log(name === 'routeEvent');
});

location.replace('http://example.com#route-event/x');
Backbone.history.checkUrl();
```

- For semantic and cross browser reasons, routes will now ignore search parameters. Routes like `search?query=...&page=3` should become `search/.../3`.
- Bugfix for normalizing leading and trailing slashes in the Router definitions. Their presence (or absence) should not affect behavior.
- Router URLs now support optional parts via parentheses, without having to use a regex.

```
var Router = Backbone.Router.extend({
  routes: {
    "optional(/:item)":           "optionalItem",
    "named/optional/(y:z)":       "namedOptionalItem",
  },
  ...
});
```

Sync

- For mixed-mode APIs, `Backbone.sync` now accepts emulateHTTP and emulateJSON as inline options.

```
var Library = Backbone.Collection.extend({
    url : function() { return '/library'; }
});

var attrs = {
    title  : "The Tempest",
    author : "Bill Shakespeare",
    length : 123
  };

library = new Library;
library.create(attrs, {wait: false});

// update with just emulateHTTP
library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
  emulateHTTP: true
});

console.log(this.ajaxSettings.url === '/library/2-the-tempest'); // true
console.log(this.ajaxSettings.type === 'POST'); // true
console.log(this.ajaxSettings.contentType === 'application/json'); // true

var data = JSON.parse(this.ajaxSettings.data);
```

```
console.log(data.id === '2-the-tempest');
console.log(data.author === 'Tim Shakespeare');
console.log(data.length === 123);

// or update with just emulateJSON

library.first().save({id: '2-the-tempest', author: 'Tim Shakespeare'}, {
  emulateJSON: true
});

console.log(this.ajaxSettings.url === '/library/2-the-tempest'); // true
console.log(this.ajaxSettings.type === 'PUT'); // true
console.log(this.ajaxSettings.contentType ==='application/x-www-form-urlencoded'); // true

var data = JSON.parse(this.ajaxSettings.data.model);
console.log(data.id === '2-the-tempest');
console.log(data.author ==='Tim Shakespeare');
console.log(data.length === 123);
```

- Consolidated `"sync"` and `"error"` events within `Backbone.sync`. They are now triggered regardless of the existence of success or error callbacks.
- Added a `"request"` event to `Backbone.sync`, which triggers whenever a request begins to be made to the server. The natural complement to the `"sync"` event.

Other

- Bug fix on change where attribute comparison uses `!==` instead of `_.isEqual`.
- Bug fix where an empty response from the server on save would not call the success function.
- To improve the performance of add, `options.index` will no longer be set in the `add` event callback.
- Removed the `Backbone.wrapError` helper method. Overriding sync should work better for those particular use cases.
- To set what library Backbone uses for DOM manipulation and Ajax calls, use `Backbone.$ =` ... instead of `setDomLibrary`.
- Added a `Backbone.ajax` hook for more convenient overriding of the default use of `$.ajax`. If AJAX is too passé, set it to your preferred method for server communication.
- Validation now occurs even during `"silent"` changes. This change means that the `isValid` method has been removed. Failed validations also trigger an error, even if an error callback is specified in the options.
