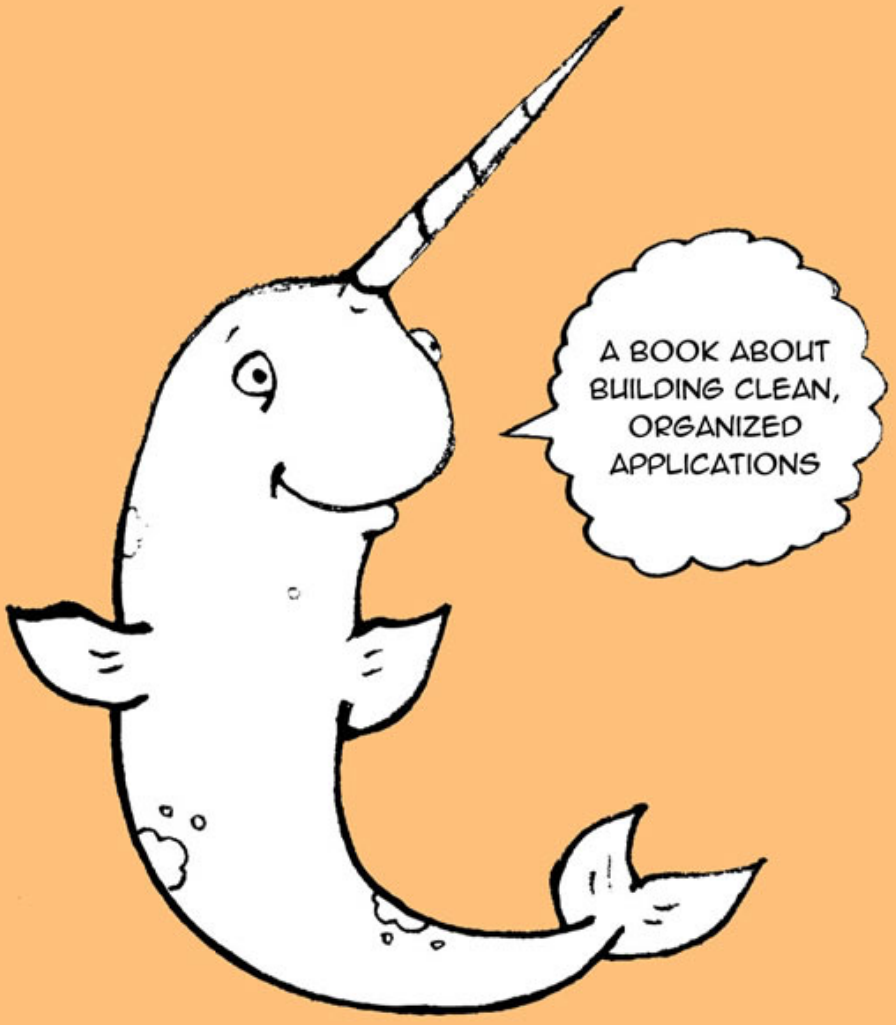
BACKBONE.js FUNDAMENTALS

ADDY OSMANI



MongoDB Ruby Driver

Getting started

Once the MongoDB Ruby driver is installed, we can begin to use it to connect to a Mongo database. To create a connection using localhost, we simply specify the driver as a dependency. Assuming we're using the default port we can then connect as follows:

```
require 'mongo'

# where 'learning-mongo' is the na
db = Connection.new.db('learning-m
```

We probably also want to place some data into 'learning-mongo'. It could be as simple as a note, so why don't we go ahead and begin a notes collection?:

`ruby notes = db.collection('notes')` Something interesting worth noting is that at this point, we haven't actually created the database nor the collection we're referencing above.

Neither of these items exist in Mongo (just yet) but as we're working with a new database but they will once we insert some real data.

A new note could be defined using key/value pairs as follows and then inserted into 'learning-mongo' using `collection.insert()`:

```
our_note = { :text => 'Remember th
note_id = notes.insert(our_note)
```

What is returned from inserting a note into the notes collection is an `ObjectId` reference for the note from Mongo. This is useful as we can re-use it to locate the same document in our database.

```
note = notes.find( :id => note_id
```

This can also be used in conjunction with Mongo's `collection.update()` method and query operators (i.e `$set`) to replace fields in an existing document.

We might update an entire document as follows:

```
note = notes.find( :id => note_id
note[:text] = 'Remember the bread
notes.update({ :_id => note_id },
```

or using `$set`, update an existing document without overwriting the entire object as like this:

```
notes.update({ :_id => note_id },
```

Useful to know: Almost each MongoDB document has an _id field as it's first attribute. This can normally be of any type, however a special BSON datatype is provided for object ids. It's a 12-byte binary value that has a high probability of being unique when allocated.

Note: Whilst we opted for the MongoDB Ruby Driver for this stack, you may also be interested in **DataMapper** - a solution which allows us to use the same API to talk to a number of different datastores. This works well for both relational and non-relational databases and more information is available on the official

project page. Sinatra: The Book also contains a brief tutorial on DataMapper for anyone interested in exploring it further.

Practical

We're going to use Sinatra in a similar manner to how we used Express in the last chapter. It will power a RESTful API supporting CRUD operations. Together with a MongoDB data store, this will allow us to easily persist data (todo items) whilst ensuring they are stored in a database. If you've read the previous chapter or have gone through any of the Todo examples covered so far, you will find this surprisingly straight-forward.

Remember that the default Todo example included with Backbone.js already persists data, although it does this via a localStorage adapter. Luckily there aren't a great deal of changes needed to switch over to using our Sinatra-based API. Let's briefly review the code that will be

powering the CRUD operations for this sections practical, as we go course won't be starting off with a near-complete base for most of our real world applications.

Installing The Prerequisites

Ruby

If using OSX or Linux, Ruby may be one of a number of open-source packages that come pre-installed and you can skip over to the next paragraph. In case you would like to check if check if you have Ruby installed, open up the terminal prompt and type:

```
$ ruby -v
```

The output of this will either be the version of Ruby installed or an error complaining that Ruby wasn't found.

Should you need to install Ruby manually (e.g for an operating system such as Windows), you can do so by downloading the latest version from http://www.ruby-lang.org/en/downloads/. Alternatively, (RVM)[http://beginrescueend.com/rvm/install/] (Ruby Version Manager) is a command-line tool that allows you to easily install and manage multiple ruby environments with ease.

Ruby Gems

Next, we will need to install Ruby Gems. Gems are a standard way to package programs or libraries written in Ruby and with Ruby Gems it's possible to install additional dependencies for Ruby applications very easily.

On OSX, Linux or Windows go to http://rubyforge.org/projects/rubygems

and download the latest version of Ruby Gems. Once downloaded, open up a terminal, navigate to the folder where this resides and enter:

```
$> tar xzvf rubygems.tgz
$> cd rubygems
$> sudo ruby setup.rb
```

There will likely be a version number included in your download and you should make sure to include this when tying the above. Finally, a symlink (symbolic link) to tie everything togther should be fun as follows:

```
$ sudo ln -s /usr/bin/gem1.8.17
/usr/bin/gem
```

To check that Ruby Gems has been correctly installed, type the following into your terminal:

```
$ gem -v
```

Sinatra

With Ruby Gems setup, we can now easily install Sinatra. For Linux or OSX type this in your terminal:

```
$ sudo gem install sinatra
```

and if you're on Windows, enter the following at a command prompt:

```
c:\\ > gem install sinatra
```

Haml

As with other DSLs and frameworks, Sinatra supports a wide range of different templating engines. ERB is the one most often recommended by the Sinatra camp, however as a part of this chapter, we're

going to explore the use of Haml to define our application templates.

Haml stands for HTML Abstractional Markup Language and is a lightweight markup language abstraction that can be used to describe HTML without the need to use traditional markup language semantics (such as opening and closing tags).

Installing Haml can be done in just a line using Ruby Gems as follows:

```
$ gem install haml
```

MongoDB

If you haven't already downloaded and installed MongoDB from an earlier chapter, please do so now. With Ruby Gems, Mongo can be installed in just one line:

```
$ gem install mongodb
```

We now require two further steps to get everything up and running.

1.Data directories

MongoDB stores data in the bin/data/db folder but won't actually create this directory for you. Navigate to where you've downloaded and extracted Mongo and run the following from terminal:

```
sudo mkdir -p /data/db/
sudo chown `id -u` /data/db
```

2.Running and connecting to your server

Once this is done, open up two terminal windows.

In the first, cd to your MongoDB bin directory or type in the complete path to it. You'll need to start mongod.

```
$ ./bin/mongod
```

Finally, in the second terminal, start the mongo shell which will connect up to localhost by default.

```
$ ./bin/mongo
```

MongoDB Ruby Driver

As we'll be using the MongoDB Ruby Driver, we'll also require the following gems:

The gem for the driver itself:

```
$ gem install mongo
```

and the driver's other prerequisite, bson:

```
$ gem install bson_ext
```

This is basically a collection of extensions used to increase serialization speed.

That's it for our prerequisites!.

Tutorial

To get started, let's get a local copy of the practical application working on our system.

Application Files

Clone this repository and navigate to `/practicals/stacks/option3`. Now run the following lines at the terminal:

```
ruby app.rb
```

Finally, navigate to `http://localhost:4567/todo` to see the application running successfully.

Note: The Haml layout files for Option 3 can be found in the /views folder.

The directory structure for our practical application is as follows:

```
--public
----css
----img
----js
-----script.js
----test
--views
app.rb
```

The `public` directory contains the scripts and stylesheets for our application and uses HTML5 Boilerplate as a base. You can find the Models, Views and Collections for this section within `public/js/scripts.js` (however, this can of course be expanded into sub-directories for each component if desired).

`scripts.js` contains the following Backbone component definitions:

```
--Models
----Todo

--Collections
----TodoList

--Views
---TodoView
---AppView
```

`app.rb` is the small Sinatra application that powers our backend API.

Lastly, the `views` directory hosts the Haml source files for our application's index and templates, both of which are compiled to standard HTML markup at runtime.

These can be viewed along with other note-worthy snippets of code from the application below.

Backbone

Views

In our main application view (AppView), we want to load any previously stored Todo items in our Mongo database when the view initializes. This is done below with the line `Todos.fetch()` in the

`initialize()` method where we also bind to the relevant events on the `Todos` collection for when items are added or changed.

```
// Our overall **AppView** is the
var AppView = Backbone.View.exten

    // Instead of generating a ne
    // the App already present in
    el: $("#todoapp"),

    // Our template for the line
    statsTemplate: _.template($('

    // Delegated events for creat
    events: {
      "keypress #new-todo":  "crea
      "keyup #new-todo":     "show
      "click .todo-clear a": "clea
    },
```

```
    // At initialization
    initialize: function() {
      this.input    = this.$("#new

      Todos.on('add',   this.addOn
      Todos.on('reset', this.addAl
      Todos.on('all',   this.rende

      Todos.fetch();
    },

    // Re-rendering the App just i
    // of the app doesn't change.
    render: function() {
      this.$('#todo-stats').html(t
        total:      Todos.length,
        done:
```

….

Collections

In the TodoList collection below, we've set the `url` property to point to `/api/todos` to reference the collection's location on the server. When we attempt to access this from our Sinatra-backed API, it should return a list of all the Todo items that have been previously stored in Mongo.

For the sake of thoroughness, our API will also support returning the data for a specific Todo item via `/api/todos/itemID`. We'll take a look at this again when writing the Ruby code powering our backend.

```
// Todo Collection

var TodoList = Backbone.Collecti

    // Reference to this collectio
```

```
  model: Todo,

  // Save all of the todo items
  // localStorage: new Store("t
  url: '/api/todos',

  // Filter down the list of al
  done: function() {
    return this.filter(function
  },

  // Filter down the list to on
  remaining: function() {
    return this.without.apply(t
  },

  // We keep the Todos in seque
  // GUID in the database. This
  nextOrder: function() {
    if (!this.length) return 1;
    return this.last().get('orde
  },
```

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

});
```

Model

The model for our Todo application remains largely unchanged from the versions previously covered in this book. It is however worth noting that calling the function `model.url()` within the below would return the relative URL where a specific Todo item could be located on the server.

```
// Our basic **Todo** model has
var Todo = Backbone.Model.extend
  idAttribute: "_id",
```

```
    // Default attributes for a t
    defaults: function() {
      return {
        done:  false,
        order: Todos.nextOrder()
      };
    },

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

Ruby/Sinatra

Now that we've defined our main models, views and collections let's get the CRUD operations required by our Backbone application supported in our Sinatra API.

We want to make sure that for any operations changing underlying data (create, update, delete) that our Mongo data store correctly reflects these.

app.rb

For `app.rb`, we first define the dependencies required by our application. These include Sinatra, Ruby Gems, the MongoDB Ruby driver and the JSON gem.

```
require 'rubygems'
require 'sinatra'
require 'mongo'
require 'json'
```

Next, we create a new connection to Mongo, specifying any custom configuration desired. If running a multi-threaded application, setting the 'pool_size' allows us to specify a maximum pool size and

'timeout' a maximum timeout for waiting for old connections to be released to the pool.

```
DB = Mongo::Connection.new.db("myc
```

Finally we define the routes to be supported by our API. Note that in the first two blocks - one for our application root (/) and the other for our todo items route `/todo` - we're using Haml for template rendering.

```
class TodoApp < Sinatra::Base

    get '/' do
      haml :index, :attr_wrapper =
    end

    get '/todo' do
      haml :todo, :attr_wrapper =>
    end
```

`haml :index` instructs Sinatra to use the `views/index.haml` for the application index, whilst `` `attr_wrapper `` is simply defining the values to be used for any local variables defined inside the template. This similarly applies Todo items with the template `views/todo.haml'.

The rest of our routes make use of the `params` hash and a number of useful helper methods included with the MongoDB Ruby driver. For more details on these, please read the comments I've made inline below:

```
get '/api/:thing' do
  # query a collection :thing, co
  # to a string representation of
  DB.collection(params[:thing]).fi
end

get '/api/:thing/:id' do
```

```
  # get the first document with t
  # than a Cursor, the standard o
  # ID conversion and the final o
  from_bson_id(DB.collection(para
end

post '/api/:thing' do
  # parse the post body of the co
  # the collection #thing and ret
  oid = DB.collection(params[:thi
  "{\"_id\": \"#{oid.to_s}\"}"
end

delete '/api/:thing/:id' do
  # remove the item with id :id f
  # representation of the object
  DB.collection(params[:thing]).re
end

put '/api/:thing/:id' do
  # collection.update() when used
  # in this case, the put request
```

```
  DB.collection(params[:thing]).up
end

# utilities for generating/conver
def to_bson_id(id) BSON::ObjectId
def from_bson_id(obj) obj.merge({

end
```

That's it. The above is extremely lean for an entire API, but does allow us to read and write data to support the functionality required by our client-side application.

For more on what MongoDB and the MongoDB Ruby driver are capable of, please do feel free to read their documentation for more information.

If you're a developer wishing to take this example further, why not try to add some additional capabilities to the service:

- Validation: improved validation of data in the API. What more could be done to ensure data sanitization?
- Search: search or filter down Todo items based on a set of keywords or within a certain date range
- Pagination: only return the Nth number of Todo items or items from a start and end-point

Haml/Templates

Finally, we move on to the Haml files that define our application index (layout.haml) and the template for a specific Todo item (todo.haml). Both of these are largely self-explanatory, but it's useful to see the differences between the Jade approach we reviewed in the last chapter vs. using Haml for this implementation.

Note: In our Haml snippets below, the forward slash character is used to indicate a comment. When this character is placed at the beginning of a line, it wraps all of the text after it into a HTML comment. e.g

```
/ These are templates
```

compiles to:

```
<!-- These are templates -->
```

index.haml

```
%head
  %meta{'charset' => 'utf-8'}/
  %title=title
  %meta{'name' => 'description',
  %meta{'name' => 'author', 'conte
  %meta{'name' => 'viewport', 'co

  / CSS concatenated and minified
```

```
    %link{'rel' => 'stylesheet', 'h
    / end CSS

    %script{'src' => 'js/libs/moder
%body
    %div#container
      %header
      %div#main
        = yield
      %footer
    /! end of #container

    %script{'src' => 'http://ajax.g

    / scripts concatenated and mini
    %script{'src' => 'js/mylibs/und
    %script{'src' => 'js/mylibs/bac
    %script{'defer' => true, 'src' =
    %script{'defer' => true, 'src' =
    / end scripts
```

todo.haml

```
%div#todoapp
  %div.title
    %h1
      Todos
      %div.content
        %div#create-todo
          %input#new-todo{"placeho
          %span.ui-tooltip-top{"s
        %div#todos
          %ul#todo-list
        %div#todo-stats

/ Templates

%script#item-template{"type" => "
  <div class="todo <%= done ? 'do
  %div.display
    <input class="check" type="che
    %div.todo-text
    %span#todo-destroy
```

```
    %div.edit
      %input.todo-input{"type" => "
    </div>

  %script#stats-template{"type" => "
    <% if (total) { %>
    %span.todo-count
      %span.number <%= remaining %>
      %span.word <%= remaining == 1
      left.
    <% } %>
    <% if (done) { %>
    %span.todo-clear
      %a{"href" => "#"}
        Clear
        %span.number-done <%= done
        completed
        %span.word-done <%= done ==
    <% } %>
```

Conclusions

In this chapter, we looked at creating a Backbone application backed by an API powered by Ruby, Sinatra, Haml, MongoDB and the MongoDB driver. I personally found developing APIs with Sinatra a relatively painless experience and one which I felt was on-par with the effort required for the Node/Express implementation of the same application.

This section is by no means the most comprehensive guide on building complex apps using all of the items in this particular stack. I do however hope it was an introduction sufficient enough to help you decide on what stack to try out for your next project.

Advanced

Modular JavaScript

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.

Organizing modules with RequireJS and AMD

In case you haven't used it before, 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.

So, why use RequireJS with Backbone? Although Backbone is excellent when it comes to providing a sanitary structure to your applications, there are a few key areas where some additional help could be used:

1. Backbone doesn't endorse a particular approach to modular-development. Although this means it's quite open-ended for developers to opt for classical patterns like the module-pattern or Object Literals for structuring their apps (which both work fine), it also means developers aren't sure of what works best when other concerns come into play, such as dependency management.

RequireJS is compatible with the AMD (Asynchronous Module Definition) format, a format which was born from a desire to write something better than the 'write lots of script tags with implicit dependencies and manage them manually' approach to development. In addition to allowing you to clearly declare dependencies, AMD works well in the browser, supports string IDs for dependencies, declaring multiple modules in the same file and gives you easy-to-use tools to avoid polluting the global namespace.

2. Let's discuss dependency management a little more as it can actually be quite challenging to get right if you're doing it by hand. When we write modules in JavaScript, we ideally want to be able to handle the reuse of code units intelligently and sometimes

this will mean pulling in other modules at run-time whilst at other times you may want to do this dynamically to avoid a large payload when the user first hits your application.

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.

I've previously written a detailed article covering both AMD and other module formats and script loaders in case you'd like to explore this topic further. The takeaway is that although it's perfectly fine to develop applications without a script loader or clean module format in place, it can be of significant benefit to consider using these tools in your application development.

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 /*functio
);
```

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 he

define(['foo', 'bar'],
    // module definition function
    // dependencies (foo and bar)
    function ( foo, bar ) {
        // return a value that de
        // (i.e the functionality

        // create your module here
        var myModule = {
            doStuff:function(){
```

```
                console.log('Yay!
            }
        }

        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 he

define(function(require){
        // module definition funct
```

```
    // dependencies (foo and bar)
    var foo = require('foo'),
        bar = require('bar');

        // return a value that de
        // (i.e the functionality

        // create your module here
        var myModule = {
            doStuff:function(){
                console.log('Yay!
            }
        }

        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 t
// In this example, the 'exports'
// function arguments to the call
// so that they can similarly be

require(['foo', 'bar'], function
        // rest of your code here
        foo.doSomething();
});
```

Wrapping modules, views and other components with AMD

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 considered it's

dependencies and so, to write an AMD model module, we would simply do this:

```
define(['underscore', 'backbone'],
  var myModel = Backbone.Model.ext

    // Default attributes
    defaults: {
      content: "hello world",
    },

    // A dummy initialization meth
    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, myC

    var AppView = Backbone.View.exte
    ...
```

Aliasing to the dollar-sign (`$`), once again makes it very easy to encapsulate any part of an application you wish using AMD.

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. Any of the other scripts being loaded are just there for the sake of example.

```
require.config( {
    paths: {
        'backbone':         'libs/
        'underscore':       'libs/
        'text':             'libs/
```

```
            'jquery':                'libs
            'json2':                 'libs
            'datepicker':            'libs
            'datepickermobile': 'libs
            'jquerymobile':          'libs
        },
        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!template
    function(app, mainView){
        // the contents of the ma
        // loaded into mainView f
    }
);
```

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="
    <% _.each( person, function( 
        <li><%= person_item.get("
    <% }); %>
</script>
```

JS:

```
var compiled_template = _.templat
```

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
    function(app, mainView){

        var compiled_template = _
    }
);
```

That's it!. You can then go applying your template to a view in Backbone doing something like:

```
collection.someview.el.html( compi
```

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 Require.js 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.

Note: If you haven't looked at build processes before and this is your first time hearing about them, you might find my post and screencast on this topic useful.

With some other structural JavaScript frameworks, my recommendation would normally be to implicitly use YUI Compressor or Google's closure compiler tools, but we have a slightly more elegant method available, when it comes to Backbone if you're using RequireJS. RequireJS has a

command line optimization tool called r.js which 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','unde
    function($,Backbone, _, Sample
        //...
    });
```

will combine fine, however inline dependencies such as:

```
var models = someCondition ? ['mo
```

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/
        'underscore':         'libs/
        'jquery':             'libs/
        'json2':              'libs/
        'datepicker':         'libs/
        'datepickermobile': 'libs/
        'jquerymobile':       'libs/
    },
    optimize: "uglify",
    modules: [
        {
            name: "app",
            exclude: [
                // If you prefer
            ]
        }
```

```
    ]
})
```

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.

Practical: 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" pla
            <span class="ui-tooltip-
          </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-m
    underscore: 'libs/underscore/
```

```
    backbone: 'libs/backbone/backb
    text: 'libs/require/text'
  }
});

require(['views/app'], function(Ap
  var app_view = new AppView;
});
```

The `require()` at the end of our main.js file is simply there so we can load and instantiation the primary view for our application (`views/app.js`). You'll commonly see both this and the configuration object included the most top-level script file 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.exte
```

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

    // Cache the template functio
    template: _.template($('#item-

    // The DOM events specific to
    events: {
      "click .check"
      "dblclick div.todo-content"
      "click span.todo-destroy"
      "keypress .todo-input"
    },

    // The TodoView listens for c
    // a one-to-one correspondence
    // app, we set a direct refer
    initialize: function() {
      this.model.bind('change', t
      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="
      <div class="todo <%= done ?
        <div class="display">
          <input class="check" typ
          <div class="todo-conten
          <span class="todo-destr
        </div>
        <div class="edit">
          <input class="todo-inpu
        </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/todos.js

```
define([
  'jquery',
  'underscore',
  'backbone',
  'text!templates/todos.html'
  ], function($, _, Backbone, todo
  var TodoView = Backbone.View.ext

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

    // Cache the template functio
    template: _.template(todosTemp

    // The DOM events specific to
    events: {
      "click .check"
      "dblclick div.todo-content"
      "click span.todo-destroy"
      "keypress .todo-input"
    },
```

```
    // The TodoView listens for c
    // a one-to-one correspondenc
    // app, we set a direct refer
    initialize: function() {
      this.model.bind('change', t
      this.model.view = this;
    },

    // Re-render the contents of
    render: function() {
      $(this.el).html(this.templa
      this.setContent();
      return this;
    },

    // Use `jQuery.text` to set t
    setContent: function() {
      var content = this.model.ge
      this.$('.todo-content').tex
      this.input = this.$('.todo-
      this.input.bind('blur', thi
      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="
      <div class="todo-content"></
      <span class="todo-destroy">
    </div>
```

```
    <div class="edit">
      <input class="todo-input" ty
    </div>
</div>
```

There's no longer a need to be concerned with IDs for the template as we can map it's 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'],
  var TodoModel = Backbone.Model.e

    // Default attributes for the
    defaults: {
      // Ensure that each todo cre
      content: "empty todo...",
      done: false
    },

    initialize: function() {
    },

    // Toggle the `done` state of
    toggle: function() {
      this.save({done: !this.get('
    },

    // Remove this Todo from *loca
    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,

  var TodosCollection = Backbone

  // Reference to this collecti
  model: Todo,

  // Save all of the todo items
  localStorage: new Store("todos

  // Filter down the list of al
  done: function() {
    return this.filter(function
  },

  // Filter down the list to on
  remaining: function() {
    return this.without.apply(th
```

```
    },
    ...
```

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/todos',
  'text!templates/stats.html'
```

```
], function($, _, Backbone, Todo

var AppView = Backbone.View.exte

  // Instead of generating a ne
  // the App already present in
  el: $("#todoapp"),

  // Our template for the line
  statsTemplate: _.template(sta

  // ...events, initialize() et

  // Re-rendering the App just
  // of the app doesn't change.
  render: function() {
    var done = Todos.done().len
    this.$('#todo-stats').html(
      total:      Todos.length,
      done:       Todos.done().
      remaining:  Todos.remainin
    }));
```

```
    },
    ...
```

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"><%=
          <span class="word"><%= 
        </span>
      <% } %>
      <% if (done) { %>
        <span class="todo-clear">
          <a href="#">
            Clear <span class="nu
            completed <span class=
          </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.

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.

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 - which Nicholas Zakas has spoken about) and AOL for some of our upcoming projects.

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.

For ease of reference, I sometimes refer to these three patterns grouped together as Aura (a word that means subtle, luminous light).

Practical

For the practical section of this chapter, we'll be extending the well-known Backbone Todo application using the three patterns mentioned above. The complete code for this section can be found here:

https://github.com/addyosmani/backbone-aura and should ideally be run on at minimum, a local HTTP server.

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 it's 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 (channe
    if (!channels[channel]) channe
    channels[channel].push(subscri
```

```
  };

  obj.publish = function (channel
    if (!channels[channel]) retur
    var args = [].slice.call(argu
    for (var i = 0, l = channels[
      channels[channel][i].apply(
    }
  };

  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
        },
        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, runn
```

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" , "..
```

```
    var facade = facade || {};
```

```
    facade.subscribe = function(su

        // Note: Handling permiss
        // The permissions check
        // to just use the mediat

        if(permissions.validate(su
            mediator.subscribe( ch
        }
    }

    facade.publish = function(char
        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 ca
    // against subscriptions prio
    // to clear. This enforces a
    // layer for your application

    var permissions = {

        newContentAvailable: {
            contentUpdater:true
        },

        endContentEditing:{
            todoSaver:true
        },

        beginContentEditing:{
            editFocus:true
        },
```

```
        addingNewTodo:{
            todoTooltip:true
        },

        clearContent:{
            garbageCollector:true
        },

        renderDone:{
            todoCounter:true //sw
        },

        destroyContent:{
            todoRemover:true
        },

        createWhenEntered:{
            keyboardManager:true
        }

};
```

```
    permissions.validate = functio
        var test = permissions[cha
        return test===undefined? 
    };

    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 - it's 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", "a
function ($, _, facade) {

    // Subscription 'modules' for
    // the form facade.subscribe(

    // Update view with latest to
    // Subscribes to: newContentA

    facade.subscribe('contentUpda
        var content = context.mode
        context.$('.todo-content')
```

```
        context.input = context.$
        context.input.bind('blur',
        context.input.val(content)
    });


    // Save models when a user ha
    // Subscribes to: endContentE
    facade.subscribe('todoSaver',
        try {
            context.model.save({
                content: context.
            });
            $(context.el).removeC
        } catch (e) {
            //console.log(e);
        }
    });


    // Delete a todo when the use
    // Subscribes to: destroyConte
```

```
    facade.subscribe('todoRemover
        try {
            context.model.clear()
        } catch (e) {
            //console.log(e);
        }
    });


    // When a user is adding a ne
    // Subscribes to: addingNewTo
    facade.subscribe('todoTooltip
        var tooltip = context.$("
        var val = context.input.va
        tooltip.fadeOut();
        if (context.tooltipTimeou
        if (val == '' || val == co
        var show = function () {
                tooltip.show().fa
            };
        context.tooltipTimeout =
    });
```

```
// Update editing UI on switc
// Subscribes to: beginConten
facade.subscribe('editFocus',
    $(context.el).addClass("e
    context.input.focus();
});


// Create a new todo entry
// Subscribes to: createWhenE
facade.subscribe('keyboardMana
    if (e.keyCode != 13) retu
    todos.create(context.newA
    context.input.val('');
});


// A Todo and remaining entry
// Subscribes to: renderDone
```

```
    facade.subscribe('todoCounter
        var done = Todos.done().le
        context.$('#todo-stats').h
            total: Todos.length,
            done: Todos.done().len
            remaining: Todos.remai
        }));
    });


    // Clear all completed todos
    // Subscribes to: clearConten
    facade.subscribe('garbageColle
        _.each(Todos.done(), funct
            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 esponse 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:

Backbone.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)

Downloads And Source Code

You can either download the raw source code for the project, fork the repository or use one of these links:

- Production: production
- Development: development version
- Examples + Source : zipball
- Repositoryhttp://github.com/addyosmani/backbone.paginator)

Live Examples

Live previews of both pagination components using the Netflix API can be found

below. Download the tarball or fork the repository to experiment with these examples further.

Demo 1: Backbone.Paginator.requestPager()

Demo 2: Backbone.Paginator.clientPager()

Demo 3: Infinite Pagination (Backbone.Paginator.requestPager())

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
```

2: Set the model and base URL 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,
          url: 'http://odata.netfli
```

3. Map the attributes supported by your API (URL)

Next, we're going to map the request (URL) parameters supported by your API or backend data service back to attributes that are internally used by Backbone.Paginator.

For example: the NetFlix API refers to it's parameter for stating how many results to skip ahead by as `$skip` and it's number of items to return per page as `$top` (amongst others). We determine these by looking at a sample URL pointing at the service:

```
http://odata.netflix.com/v2/Catal
```

We then simply map these parameters to the relevant Paginator equivalents shown on the left hand side of the next snippets to get everything working:

```
// @param-name for the qu
// request (e.g query/key
queryAttribute: '$filter',

// @param-name for number
perPageAttribute: '$top',

// @param-name for how ma
skipAttribute: '$skip',

// @param-name for the di
sortAttribute: '$sort',

// @param-name for field
orderAttribute: '$orderBy
```

```
        // @param-name for the fo
        formatAttribute: '$format

        // @param-name for a custo
        customAttribute1: '$inline

        // @param-name for anothe
        customAttribute2: '$callba
```

Note: you can define support for new custom attributes in Backbone.Paginator if needed (e.g customAttribute1) for those that may be unique to your service.

4. Configure the default pagination, query and sort details for the paginator

Now, let's configure the default values in our collection for these parameters so that as a user navigates through the paginated

UI, requests are able to continue querying with the correct field to sort on, the right number of items to return per request etc.

e.g: If we want to request the:

- 1st page of results
- for the search query 'superman'
- in JSON format
- sorted by release year
- in ascending order
- where only 30 results are returned per request

This would look as follows:

```
// current page to query
page: 5,

// The lowest page index
firstPage: 0, //some begi
```

```
        // how many results to qu
        // per request)
        perPage: 30,

        // maximum number of page
        // the server (only here
        // service doesn't return
        totalPages: 10,

        // what field should the
        sortField: 'ReleaseYear',

        // what direction should
        sortDirection: 'asc',

        // what would you like to
        // as Netflix reqires add
        // we simply fill these a
        query: "substringof('" + e

        // what format would you
        format: 'json',
```

```
            // what other custom para
            // you require
            // for your application?
            customParam1: 'allpages',

            customParam2: 'callback',
```

As the particular API we're using requires `callback` and `allpages` parameters to also be passed, we simply define the values for these as custom parameters which can be mapped back to requestPager as needed.

5. 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
                    // are structured (e.
                    var tags = response.d
                    //Normally this.total
                    //but as this particu
                    //total count of item
                    this.totalPages = Mat
```

```
                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)** - go to a specific page
- **Collection.requestNextPage()** - go to the next page
- **Collection.requestPreviousPage()** - go to the previous page

- **Collection.howManyPer(n)** - set the number of items to display per page

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 â€˜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 and base URL:

```
var PaginatedCollection = Backbone

        model: model,

        url: 'http://odata.netfli
```

2. Map the attributes supported by your API (URL)

We're similarly going to map request parameter names for your API to those supported in the paginator:

```
        perPageAttribute: '$top',

        skipAttribute: '$skip',

        orderAttribute: '$orderBy
```

```
customAttribute1: '$inline

queryAttribute: '$filter'

formatAttribute: '$format

customAttribute2: '$callba
```

3. Configure how to paginate data at a UI-level

We then get to configuration for the paginated data in the UI. `perPage` specifies how many results to return from the server whilst `displayPerPage` configures how many of the items in returned results to display per 'page' in the UI. e.g If we request 100 results and only display 20 per page, we have 5 sub-pages of results that can be navigated through in the UI.

```
        // M: how many results to
        perPage: 40,

        // N: how many results to
        // Effectively M/N = the
        displayPerPage: 20,
```

4. Configure the rest of the request parameter default values

We can then configure default values for the rest of our request parameters:

```
        // current page to query
        page: 1,

        // a default. This should
        // sort direction
        sortDirection: 'asc',

        // sort field
        sortField: 'ReleaseYear',
```

```
        //or year(Instant/Availab

        // query
        query: "substringof('" + e

        // request format
        format: 'json',

        // custom parameters for
        // application
        customParam1: 'allpages',

        customParam2: '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
            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.pager(sortBy, sortDirection)** - update sort on the current view

Views/Templates

Although the collection layer is perhaps the most important part of Backbone.Paginator, it would be of little use without views interacting with it. The project zipball comes with three complete examples of using the components with the Netflix API, but here's a sample view and template from the `requestPager()` example for those interested in learning more:

First, we have a view for a pagination bar in our UI that allows us to navigate around our paginated collection:

```
(function ( views ) {

    views.PaginatedView = Backbone

        events: {
            'click a.servernext':
            'click a.serverprevio
            'click a.orderUpdate'
            'click a.serverlast':
            'click a.page': 'goto
            'click a.serverfirst'
            'click a.serverpage':
            'click .serverhowmany

        },

        tagName: 'aside',
```

```
    template: _.template($('#
    initialize: function () {
        this.collection.on('re
        this.collection.on('cl
        this.$el.appendTo('#pa
    },
    render: function () {
        var html = this.templa
        this.$el.html(html);
    },
    updateSortBy: function (e
        e.preventDefault();
        var currentSort = $('
        this.collection.update
    },
    nextResultPage: function
```

```
            e.preventDefault();
            this.collection.reques
        },

        previousResultPage: funct
            e.preventDefault();
            this.collection.reques
        },

        gotoFirst: function (e) {
            e.preventDefault();
            this.collection.goTo(
        },

        gotoLast: function (e) {
            e.preventDefault();
            this.collection.goTo(
        },

        gotoPage: function (e) {
            e.preventDefault();
            var page = $(e.target
```

```
                this.collection.goTo(
            },

            changeCount: function (e)
                e.preventDefault();
                var per = $(e.target)
                this.collection.howMa
            }

    });

})( app.views );
```

which we use with a template like this to generate the necessary pagination links (more are shown in the full example):

```
<span class="divider">/</span>
            <% if (page > firstPage)
                <a href="#" class="se
            <% }else{ %>
                <span>Previous</span>
```

```
<% }%>
<% if (page < totalPages)
    <a href="#" class="se
<% } %>
<% if (firstPage != page)
    <a href="#" class="se
<% } %>
<% if (lastPage != page)
    <a href="#" class="se
<% } %>
<span class="divider">/</s
<span class="cell serverh
    Show
    <a href="#" class="se
    |
    <a href="#" class="">
    |
    <a href="#" class="">
    per page
</span>
<span class="divider">/</s
<span class="cell first re
```

```
                Page: <span class="cu
                of
                <span class="total"><
                                shown
            </span>
<span class="divider">/</span>
    <span class="cell sort">
        <a href="#" class="orderUp
    </span>
    <select id="sortByField">
        <option value="cid">Selec
        <option value="ReleaseYea
        <option value="ShortName">
    </select>
</span>
```

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 = fa
```

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 , { trans
```

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.

Practical: A Backbone, RequireJS/AMD app with jQuery Mobile

Note: The code for this practical 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.mobi
    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

Unit Testing Backbone Applications With 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 it's 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

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 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 valu
    var counter = 0;
    counter++;
});
```

On it's 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 valu
    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', :
        ...
    });

    it('can subtract a number', fu
        ...
    });
});
```

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', :
        // the counter was = 1
        counter = counter + 1;
        expect(counter).toEqual(2)
    });

    it('can subtract a number', fu
        // 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 = funct
```

```
    return arg;
}

describe('a simple spy', function
    it('should spy on an instance
        var myTodo = new Todo();
        spyOn(myTodo, 'setComplete
        myTodo.setComplete('foo ba

        expect(myTodo.setComplete

        var myTodo2 = new Todo();
        spyOn(myTodo2, 'setComple

        expect(myTodo2.setComplete

    });
});
```

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 execute
    spyOn($, "ajax").andCallFake(:
        options.success();
    });

    var callback = jasmine.create
    getTodo(15, callback);

    expect($.ajax.mostRecentCall.a
    expect(callback).toHaveBeenCal
});

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 re

    var callback = jasmine.create
    getTodo(16, callback);

    waitsFor(function() {
        return callback.callCount
    });

    runs(function() {
        expect(callback).toHaveBe
    });
});

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 groceri
        done: false
    });
});

it("should contain a text value i
    expect(this.todo.get('text')).
});
```

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/
<script type="text/javascript" src
<script type="text/javascript" src
```

Next, some sample tests are included:

```
<script type="text/javascript" src
<script type="text/javascript" src
```

And finally the sources being tested:

```
<script type="text/javascript" src
<script type="text/javascript" src
```

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', functi
```

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 va
    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.exte

    defaults: function() {
        return {
            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
    var todo = new Todo({ text: '
```

```
        // what are the values expecte
        // attributes in our Todo?

        expect(todo.get('text')).toBe
        expect(todo.get('done')).toBe
        expect(todo.get('order')).toBe
});
```

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

    var spy = jasmine.createSpy('
```

```
    var todo = new Todo();

    // how do we monitor changes
    todo.bind('change', spy);

    // what would you need to do
    todo.set({ text: 'Get oil cha

    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

    var errorCallback = jasmine.c

    var todo = new Todo();

    todo.bind('error', errorCallba

    // What would you need to set
    // cause validation to fail?
```

```
        todo.set({done:'a non-integer
        var errorArgs = errorCallback
        expect(errorArgs).toBeDefined
        expect(errorArgs[0]).toBe(todo
        expect(errorArgs[1]).toBe('To
});
```

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
        return 'Todo.done must be
```

```
    }
}
```

If you would like to review the final code for our Todo model, you can find it below:

```
var NAUGHTY_WORDS = /crap|poop|he

function sanitize(str) {
    return str.replace(NAUGHTY_WO
}

window.Todo = Backbone.Model.exte

    defaults: function() {
        return {
            text: '',
            done:  false,
            order: 0
        };
    },
```

```
    initialize: function() {
        this.set({text: sanitize(
    },

    validate: function(attrs) {
        if (attrs.hasOwnProperty(
            return 'Todo.done mus
        }
    },

    toggle: function() {
        this.save({done: !this.ge
    }

});
```

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 it's 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', fu

    it('Can add Model instances a
        var todos = new TodoList(

        expect(todos.length).toBe

        todos.add({ text: 'Clean

        // how many todos have be
        expect(todos.length).toBe
```

```
            todos.add([
                { text: 'Do the laundi
                { text: 'Go to the gyr
            ]);

            // how many are there in
            expect(todos.length).toBe
        });
...
```

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 de
            var todos = new TodoList(

            // what has been specifie
```

```
        expect(todos.url).toBe('/
});
```

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.Colle

        model: Todo,

        url: '/todos/',

        done: function() {
            return this.filter(fu
        },

        remaining: function() {
            return this.without.a
        },

        nextOrder: function() {
            if (!this.length) {
                return 1;
            }

            return this.last().ge
        },
```

```
        comparator: function(todo
            return todo.get('orde
        }

    });
```

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"/>')).toBeC`
- `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</d
```

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
```

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).toHaveBeenTriggere` `lector)` functions. An example of usage may look as follows:

```
spyOnEvent($('#el'), 'click');
$('#el').click();
expect('click').toHaveBeenTrigger
```

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', fu

    beforeEach(function() {
        $('body').append('<ul id='
        this.todoView = new TodoV
    });
```

```
    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 eleme
    //what html element tag name
    expect(todoView.el.tagName.to
});
```

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.exte
    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"
    expect($(this.view.el)).toHaveC
});
```

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,

    expect(todoView.model).toBeDe

    // what's the value for Todo.
    expect(todoView.model.get('do
});
```

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.Mode
      text: "My Todo",
      order: 1,
      done: false
    });
    this.view = new TodoView({mode
  });

  describe("Rendering", function(

    it("returns the view object",
```

```
      expect(this.view.render()).
    });

    it("produces the correct HTML
      this.view.render();

      //let's use jasmine-jquery'
      //testing for the complete
      expect(this.view.el.innerHT
        .toContain('<label class=
    });

  });

});
```

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" typ
          <label class="todo-conte
          <span class="todo-destro
        </div>
        <div class="edit">
          <input class="todo-input
        </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="t
  var output = template
    .replace("<%= text %>", this.
  $(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 conten
    expect($(this.view.el).find('
      .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'
  });

  ...

  describe("Template", function()

    beforeEach(function() {
      $('.todos').append(this.vie
```

```
    });

    it("has the correct text conte
      expect($('.todos').find('.to
        .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", fu

  beforeEach(function() {
    this.model.set({done: true},
    $('.todos').append(this.view.
  });

  it("has a done class", function
    expect($('.todos .todo-conten
      .toHaveClass("done");
  });

});
```

This will fail with the following message:

Expected 'My Todo' to have class 'done'.

which can be fixed in the existing render() method as follows:

```
render: function() {
  var template = '<label class="t
    '<%= text %></label>';
  var output = template
    .replace("<%= text %>", this.
  $(this.el).html(output);
  if (this.model.get('done')) {
    this.$(".todo-content").addCl
  }
  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 Micro-templating. 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.exte

  tagName: "li",

  initialize: function(options) {
    this.template = _.template(op
  },

  render: function() {
    $(this.el).html(this.template
    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.
    todo: '<label class="todo-con
            '<%= 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.to
    });
  });

  ...

});
```

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.
```

```
        todo: '<label class="todo-con
                '<%= text %>' +
              '</label>'
    });
});
```

This will now also pass without any issues. Remember that jasmine-jquery also supports loading external fixtures into your specs easily using it's 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 peaking 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</titl

      <link rel="stylesheet" href="
      <script src="qunit.js"></scr

      <!-- Your application -->
      <script src="app.js"></script

      <!-- Your tests -->
      <script src="tests.js"></scr
```

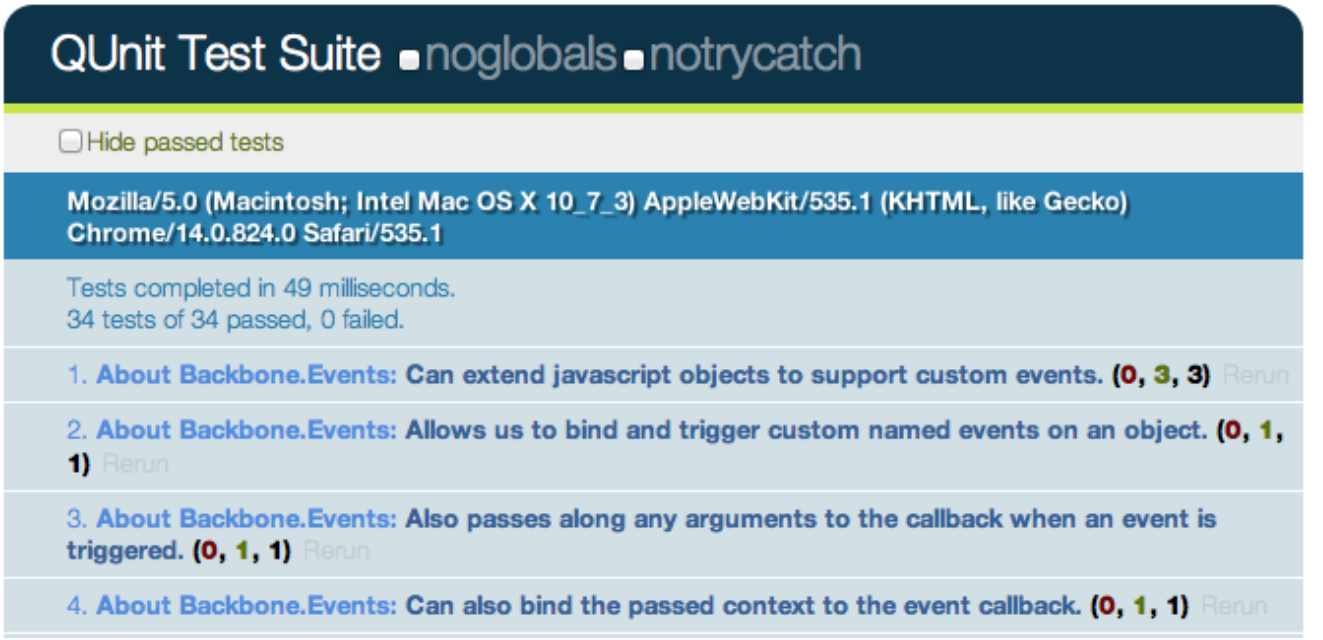
```
</head>
<body>
    <h1 id="qunit-header">QUnit Te
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-tool
    <h2 id="qunit-userAgent"></h2>
    <ol id="qunit-tests">test mark
</body>
</html>
```

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

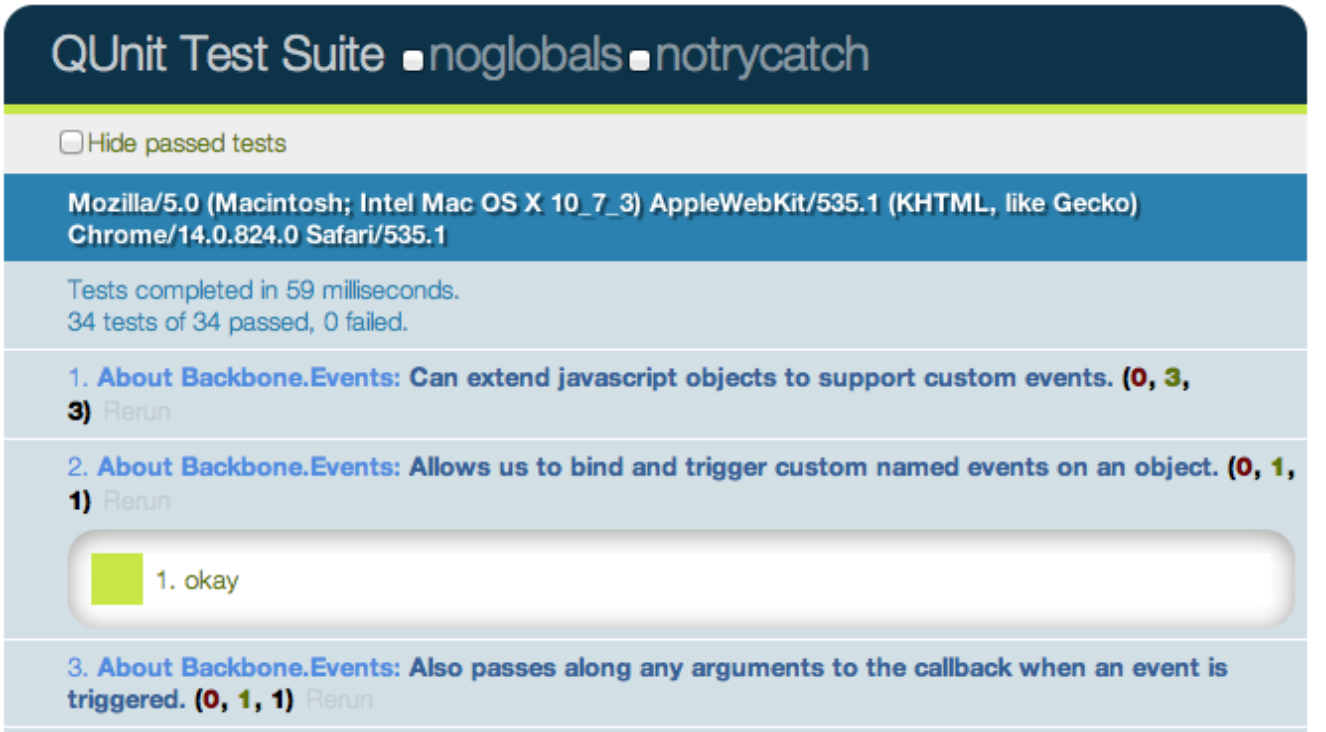
When running correctly, the above test runner looks as follows:



screenshot 1

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.



screenshot 2

If however any tests fail, the test gets highlighted (and the qunit-banner at the top switches to red):

screenshot 3

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 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 - asse

    // ok( boolean, message )
    ok( true, 'the test succeeds'
    ok( false, 'the test fails');

    // equal( actualValue, expecte
    equal( myString, 'Hello Backbo
});
```

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(
}

test( 'reverseString()', function
    expect( 5 );
    equal( reverseString('hello')
    equal( reverseString('foobar'
    equal( reverseString('world')
    notEqual( reverseString('worl
    equal( reverseString('bubble'
})
```

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 colle
// list of stores

var Store = Backbone.Model.extend

var StoreList = Backbone.Collectio
    model: store,
    comparator: function( store )
});

// Define a group for our tests
module( "StoreList sanity check",
    setup: function() {
        this.list = new StoreList(
        this.list.add(new Store({
        this.list.add(new Store({
        this.list.add(new Store({
        this.list.add(new Store({
    },
```

```
    teardown: function() {
        window.errors = null;
    }
});

// Test the order of items added
test( "test ordering", function()
    expect( 1 );
    var expected = ["Barnes & Nob
    var actual = this.list.pluck(
    deepEqual( actual, expected,
});
```

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
  equal( actual, 1,     "passes as
});
```

notEqual - a comparison assertion. It passes if actual != expected

```
test( "notEqual", 2, function() {
  var actual = 6 - 5;
  notEqual( actual, false, "passes
  notEqual( actual, 0,     "passes
});
```

strictEqual - a comparison assertion. It passes if actual === expected.

```
test( "strictEqual", 2, function()
  var actual = 6 - 5;
  strictEqual( actual, true,  "fa
  strictEqual( actual, 1,     "pa
});
```

notStrictEqual - a comparison assertion. It passes if actual !== expected.

```
test("notStrictEqual", 2, functio
  var actual = 6 - 5;
  notStrictEqual( actual, true,
  notStrictEqual( actual, 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').childre

  equal( actual, {q: 'foo', t: 'ba
```

```
  deepEqual( actual, {q: 'foo', t
  equal( el, children, "fails - j
  deepEqual(el, children, "fails

});
```

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:
  notDeepEqual( actual, {q: 'foo'
});
```

raises - an assertion which tests if a callback throws any exceptions

```
test("raises", 1, function() {
  raises(function() {
    throw new Error( "Oh no! It's
  }, "passes - an error was throw
});
```

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="
    <script src="qunit.js"></scri
    <script src="app.js"></script
    <script src="tests.js"></scri
</head>
<body>
    <h1 id="qunit-header">QUnit Te
    <h2 id="qunit-banner"></h2>
    <div id="qunit-testrunner-too
    <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 !== "unde
          // Since `start` value wa
          // the initial jQuery obj

          return this.each(function
            $(this).prepend( "<b>" +
          });

        } else {
```

```
            // Since no `start` value
            // getter, returing the a
            // selected element.

            var val = this.eq( 0 ).ch
            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&qu
    &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 `$(el).enumerate()`
- **o passed as an argument** - i.e `$(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, fu
  var items = $("#qunit-fixture li
  equal( items.eq(0).text(), "1. I
  equal( items.eq(1).text(), "2. w
  equal( items.eq(2).text(), "3. i
  equal( items.eq(3).text(), "4. a
  equal( items.eq(4).text(), "5. i
});
```

```
test( "0 passed as an argument", 5
  var items = $("#qunit-fixture li
  equal( items.eq(0).text(), "0. P
  equal( items.eq(1).text(), "1. w
  equal( items.eq(2).text(), "2. i
  equal( items.eq(3).text(), "3. a
  equal( items.eq(4).text(), "4. i
});

test( "1 passed as an argument", 5
  var items = $("#qunit-fixture li
  equal( items.eq(0).text(), "1. P
  equal( items.eq(1).text(), "2. w
  equal( items.eq(2).text(), "3. i
  equal( items.eq(3).text(), "4. a
  equal( items.eq(4).text(), "5. i
});
```

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
    var message = getUniqueString
    var spy = this.spy();

    PubSub.subscribe( message, spy
    PubSub.publishSync( message, "

    ok( spy1.calledOnce, "the subs
});
```

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 jQuery.getJS
    this.spy( jQuery, "ajax" );

    jQuery.getJSON( "/todos/comple

    ok( jQuery.ajax.calledOnce );
    equals( jQuery.ajax.getCall(0)
    equals( jQuery.ajax.getCall(0)
});
```

SinonJS comes with a rich spy interface which callows 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 here (http://sinonjs.org/docs/), 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 wi
    var spy = sinon.spy();

    PubSub.subscribe( "message", s
    PubSub.publishSync( "message",

    assertTrue( spy.calledWith( {
});
```

Stricter argument matching: test a spy was called at least once with specific arguments and no others:

```
test( "Should call a subscriber wi
    var spy = sinon.spy();

    PubSub.subscribe( "message", s
    PubSub.publishSync( "message"
    PubSub.publishSync( "message"

    // This passes
    assertTrue( spy.calledWith("ma

    // This however, fails
    assertTrue( spy.calledWithExac
});
```

Testing call order: testing if a spy was called before or after another spy:

```
test( "Should call a subscriber ar
    var a = sinon.spy();
    var b = sinon.spy();

    PubSub.subscribe( "message", a
    PubSub.subscribe( "event", b )

    PubSub.publishSync( "message"
    PubSub.publishSync( "event",

    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 ar
    var message = getUniqueString
    var spy = this.spy();

    PubSub.subscribe( message, spy
    PubSub.publishSync( message, "

    // Passes if spy was called or
    ok( spy.calledOnce ); // calle

    // The number of recorded cal
    equal( spy.callCount, 1 );

    // Directly checking the argu
    equals( spy.getCall(0).args[0]
});
```

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.Collectio
    model: Todo
});

// Let's assume our instance of t
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( windo
```

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.todoStub = sinon.stub( w
```

```
        this.model = new Backbone.Mode
          id: 2,
          title: "Hello world"
        });
        this.todoStub.returns( this.m
});
```

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 inst

  setup:function() {

    this.todoStub = sinon.stub(win
    this.model = new Backbone.Mode
      id: 2,
      title: "Hello world"
    });

    this.todoStub.returns(this.mod
    this.todos = new TodoList();

    // Let's reset the relationsh
    this.todos.model = Todo;
    this.todos.add({
      id: 2,
      title: "Hello world"
    });
```

```
  },

  teardown: function() {
    this.todoStub.restore();
  }

});

test("should add a model", functi
    equal( this.todos.length, 1 )
});

test("should find a model by id",
    equal( this.todos.get(5).get('
  });
});
```

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
    var myAPI = { clearTodo: funct

    var spy = this.spy();
    var mock = this.mock( myAPI )
    mock.expects( "clearTodo" ).or

    PubSub.subscribe( "message", m
    PubSub.subscribe( "message", s
    PubSub.publishSync( "message"

    mock.verify();
```

```
    ok( spy.calledOnce );
});
```

Practical

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
    expect( 1 );

    var todo = new Todo();

    equal( todo.get('text'), "" )
});

test('Will set attributes on the m
    expect( 3 );

    var todo = new Todo( { text:

    equal( todo.get('text'), "Get
    equal( todo.get('done'), false
    equal( todo.get('order'), 0 )
});

test('Will call a custom initiali
    expect( 1 );
```

```
    var toot = new Todo({ text: 'S
    equal( toot.get('text'), 'Stop
});

test('Fires a custom event when th
    expect( 1 );

    var spy = this.spy();
    var todo = new Todo();

    todo.bind( 'change', spy );
    // How would you update a prop
    // Hint: http://documentcloud
    todo.set( { text: "new text"

    ok( spy.calledOnce, "A change
});


test('Can contain custom validatio
    expect( 3 );
```

```
    var errorCallback = this.spy()
    var todo = new Todo();

    todo.bind('error', errorCallba
    // What would you need to set
    todo.set( { done: "not a boole

    ok( errorCallback.called, 'A
    notEqual( errorCallback.getCa
    equal( errorCallback.getCall(

});
```

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.Collectio

test( 'Can add Model instances as
    expect( 3 );

    var todos = new TodoList();
    equal( todos.length, 0 );

    todos.add( { text: 'Clean the
    equal( todos.length, 1 );

    todos.add([
        { text: 'Do the laundry',
        { text: 'Go to the gym' }
    ]);
```

```
    equal( todos.length, 3 );
});

test( 'Can have a url property to
    expect( 1 );
    var todos = new TodoList();
    equal( todos.url, '/todos/' )
});

test('Fires custom named events wh
    expect(2);

    var todos = new TodoList();
    var addModelCallback = this.sp
    var removeModelCallback = this

    todos.bind( 'add', addModelCal
    todos.bind( 'remove', removeMo

    // How would you get the 'add
    todos.add( {text:"New todo"} )
```

```
    ok( addModelCallback.called )

    // How would you get the 'remo
    todos.remove( todos.last() );

    ok( removeModelCallback.calle
});
```

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='
        this.todoView = new TodoVi
    },
    teardown: function() {
        this.todoView.remove();
        $('#todoList').remove();
    }
});

test('Should be tied to a DOM elem
    expect( 1 );
    equal( this.todoView.el.tagNam
});

test('Is backed by a model instanc
```

```
    expect( 2 );
    notEqual( this.todoView.model,
    equal( this.todoView.model.ge
});

test('Can render, after which the
    this.todoView.render();

    // Hint: render() just builds
    //       How would you append
    //       How do you access the
    //
    // Hint: http://documentcloud

    $('ul#todoList').append(this.
    equal($('#todoList').find('li
});

asyncTest('Can wire up view metho
    expect( 2 );
    var viewElt;
```

```
    $('#todoList').append( this.to

    setTimeout(function() {
        viewElt = $('#todoList li

        equal(viewElt.length > 0,

        // Make sure that QUnit k
        start();
    }, 1000, 'Expected DOM Elt to


    // Hint: How would you trigge
    //       (See todos.js line 7
    //
    // Hint: http://api.jquery.co

    $('#todoList li input.check')
    expect( this.todoView.model.ge
});
```

Events

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, Backbo
```

```
        this.obj.unbind(); // rem
    }
});

test('Can extend JavaScript objec
    expect(3);

    var basicObject = {};

    // How would you give basicOb
    // Hint: http://documentcloud
    _.extend( basicObject, Backbo

    equal( typeof basicObject.bin
    equal( typeof basicObject.unb
    equal( typeof basicObject.tri
});

test('Allows us to bind and trigg
    expect( 1 );

    var callback = this.spy();
```

```
    this.obj.bind( 'basic event',
    this.obj.trigger( 'basic even

    // How would you cause the ca
    ok( callback.called );
});

test('Also passes along any argume
    expect( 1 );

    var passedArgs = [];

    this.obj.bind('some event', fu
        for (var i = 0; i < argume
            passedArgs.push( argu
        }
    });

    this.obj.trigger( 'some event

    deepEqual( passedArgs, ['arg1
```

```
});


test('Can also bind the passed co
    expect( 1 );

    var foo = { color: 'blue' };
    var changeColor = function()
        this.color = 'red';
    };

    // How would you get 'this.co
    this.obj.bind( 'an event', cha
    this.obj.trigger( 'an event' )

    equal( foo.color, 'red' );
});

test( "Uses 'all' as a special eve
    expect( 2 );

    var callback = this.spy();
```

```
    this.obj.bind( 'all', callbac
    this.obj.trigger( "custom ever
    this.obj.trigger( "custom ever

    equal( callback.callCount, 2 )
    equal( callback.getCall(0).arg
});

test('Also can remove custom event
    expect( 5 );

    var spy1 = this.spy();
    var spy2 = this.spy();
    var spy3 = this.spy();

    this.obj.bind( 'foo', spy1 );
    this.obj.bind( 'bar', spy1 );
    this.obj.bind( 'foo', spy2 );
    this.obj.bind( 'foo', spy3 );

    // How do you unbind just a s
```

```
    this.obj.unbind( 'foo', spy1 )
    this.obj.trigger( 'foo' );

    ok( spy2.called );

    // How do you unbind all calli
    this.obj.unbind( 'foo' );
    this.obj.trigger( 'foo' );

    ok( spy2.callCount, 1 );
    ok( spy2.calledOnce, "Spy 2 ca
    ok( spy3.calledOnce, "Spy 3 ca

    // How do you unbind all calli
    this.obj.unbind( '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 Applicatio
    setup: function() {
        Backbone.localStorageDB =
        $('#qunit-fixture').appen
        this.App = new TodoApp({ a

    },

    teardown: function() {
        this.App.todos.reset();
        $('#app').remove();
    }
```

```
});

test('Should bootstrap the applica
    expect( 2 );

    notEqual( this.App.todos, unde
    equal( this.App.todos.length,
});

test( 'Should bind Collection ever
      $('#new-todo').val( 'Foo' )
      $('#new-todo').trigger(new

      equal( this.App.todos.lengt
 });
```

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

Whilst we get with Backbone out of the box can be terribly useful, there are some equally beneficial add-ons that can help simplify our development process. These include:

- Backbone Layout Manager
- Backbone Boilerplate
- Backbone Model Binding
- Backbone Relational - for model relationships
- View and model inheritance
- Backbone Marionette
- Backbone CouchDB
- Backbone Validations - HTML5 inspired validations

In time, there will be tutorials in the book covering some of these resources but until then, please feel free to check them out.

Conclusions

That's it for 'Developing Backbone.js Applications'. I hope you found this book both useful, enlightening and a good start for your journey into exploring Backbone.js.

Remember, If there are other topics or areas of this book you feel could be expanded further, please feel free to let me know, or better yet, send a pull request upstream. I'm always interested in making this title as comprehensive as possible.

Until next time, the very best of luck with the rest of your journey!
