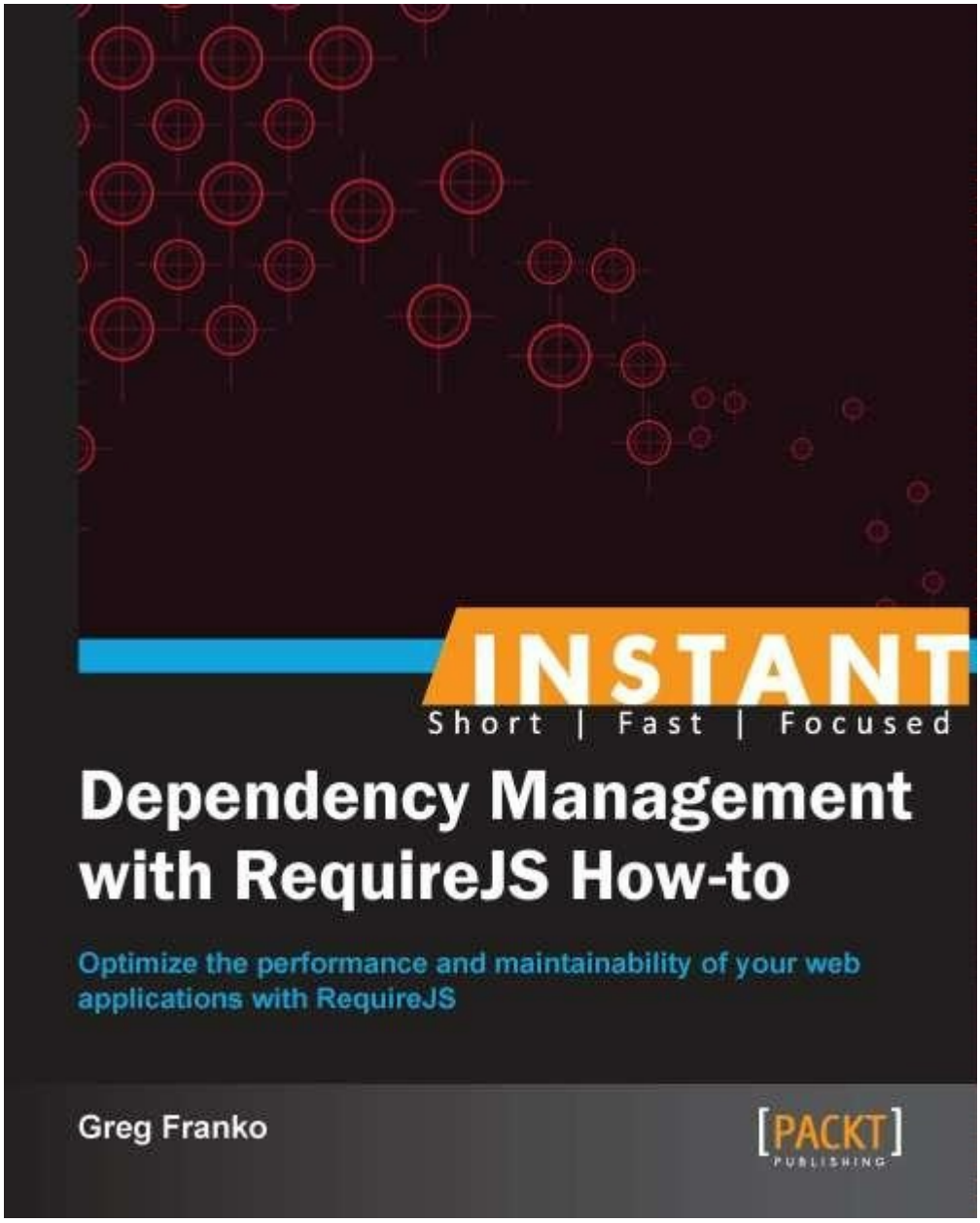
INSTANT
Short | Fast | Focused
Dependency Management with RequireJS How-to
Optimize the performance and maintainability of your web applications with RequireJS
Greg Franko
[PACKT]
PUBLISHING

Table of Contents

Instant Dependency Management with RequireJS How-to

Instant Dependency Management with RequireJS How-to

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: May 2013

Production Reference: 1140513

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78216-906-2

www.packtpub.com

Credits

Author

Greg Franko

Reviewer

Miller Medeiros

Acquisition Editor

Aarthi Kumaraswamy

Commissioning Editor

Poonam Jain

Technical Editor

Nitee Shetty

Copy Editor

Laxmi Subramanian

Project Coordinator

Joel Goveya

Proofreader

Bernadette Watkins

Production Coordinator

Nitesh Thakur

Cover Image

Conidon Miranda

About the Author

Greg Franko is a 24-year-old JavaScript engineer at AddThis and the maintainer of several popular open source JavaScript projects. He is also a technical writer who blogs regularly at http://gregfranko.com and has contributed to the open source book, *Backbone.js Fundamentals*, *O'Reilly Media*.

He is currently a lead JavaScript developer for AddThis.com and the popular AddThis third-party tool suite. He has also worked as a software engineer at AOL where he created the mobile web registration apps for AOL.com, AOL Mail, MapQuest, Aim, and The Huffington Post.

I would like to thank my mom, Janet, and my sister, Emily, for always being there to support and love me. I would also like to thank my girlfriend, Jenny, for putting up with my endless nights of coding and helping me edit this book. Without them, this book would not have been possible.

About the Reviewer

Miller Medeiros started to design and develop websites for fun in 1999 and has been doing it ever since. He is an active open source contributor; he works on projects such as js-signals, crossroads.js, mout.js, RequireJS, and syntastic.vim. He also has a blog (blog.millermedeiros.com) where he shares his knowledge and writes about conceptual topics related to programming.

I would like to thank my wife, Marina, for all the support and patience.

www.PacktPub.com

Support files, eBooks, discount offers and more

You might want to visit www.PacktPub.com for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at www.PacktPub.com and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at <service@packtpub.com> for more details.

At www.PacktPub.com, you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.

http://PacktLib.PacktPub.com

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

Free Access for Packt account holders

If you have an account with Packt at www.PacktPub.com, you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.

Preface

Welcome to *Dependency Management with RequireJS How-to*. Here, we will learn the concepts of the Asynchronous Module Definition (AMD) specification and how using Require.js, an AMD script loader built by James Burke, can improve the performance and maintainability of your web application. We will specifically learn how to integrate Require.js with popular libraries, such as jQuery, the jQueryUI Widget Factory, Backbone.js, Lodash.js, Jasmine, and more, into an AMD workflow. We will also review popular Require.js plugins and learn how to incorporate them to support common web development tasks, including dynamically rendering JavaScript templates, optimizing JavaScript files (that is, concatenating and minifying), and more.

Applications of all sizes can benefit from using Require.js. A small application can squeeze out every last kilobyte (KB) of performance using Require.js APIs for asynchronous loading and dynamic asset loading. Intermediate- to large-sized applications can become more maintainable by using the Require.js dependency management API to track file dependencies and decouple large files into smaller modules.

Let the AMD journey begin!

What this book covers

Using RequireJS (Intermediate) demonstrates how to synchronously and asynchronously include Require.js within your HTML page to kick start your application.

Setting RequireJS configurations (Intermediate) demonstrates how to set application-level Require.js parameters.

Creating AMD modules (Advanced) reviews the AMD module specification and how to register many different types of files as AMD modules.

Using jQuery and jQueryUI Widget Factory plugins with RequireJS (Advanced) demonstrates how to use jQuery and jQueryUI Widget Factory plugins with Require.js.

Using Backbone.js and Lodash.js with RequireJS (Intermediate) demonstrates how to integrate Backbone.js and Lodash.js with Require.js.

Separating mobile web and desktop logic with RequireJS (Intermediate) demonstrates techniques for maintaining one codebase to support mobile

web and desktop versions of an application using different Require.js configurations.

Using JavaScript templates with the RequireJS text plugin (Intermediate) demonstrates how to use the Require.js text plugin with Lodash JavaScript templates.

Creating Jasmine unit tests with RequireJS (Intermediate) demonstrates how to use Jasmine unit testing within an AMD environment.

Using the RequireJS optimizer (Advanced) demonstrates how to optimize an entire project with the Require.js optimizer.

What you need for this book

Most of the code examples in this book only require a text editor and a web browser. A few of the examples also require node.js (http://nodejs.org/) to be installed as a global environment variable. Many of the code examples reference snippets from Backbone-Require-Boilerplate (https://github.com/gfranko/Backbone-Require-Boilerplate), a popular open source library that I created to help developers to start using Backbone.js and Require.js together.

Who this book is for

This book is targeted at intermediate to advanced JavaScript developers who have little AMD experience.

Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text, database table names, folder names, filenames, file extensions, pathnames, dummy URLs, user input, and Twitter handles are shown as follows: "We can include external JavaScript files with the use of the `script` tag."

A block of code is set as follows:

```
var example = "this";
```

When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:

```
var example = "this";
example += " is an example";
```

New terms and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "Require.js also allows modules to be declared using the **Simplified CommonJS** wrapper syntax".

> ### Note
>
> Warnings or important notes appear in a box like this.

> ### Tip
>
> Tips and tricks appear like this.

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an email to <feedback@packtpub.com>, and mention the book title via the subject of your message.

If there is a book that you need and would like to see us publish, please send us a note in the **SUGGEST A TITLE** form on www.packtpub.com or e-mail <suggest@packtpub.com>.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on www.packtpub.com/authors.

Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.PacktPub.com. If you purchased this book elsewhere, you can visit http://www.PacktPub.com/support and register to have the files e-mailed directly to you.

Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books—maybe a mistake in the text or the code—we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting http://www.packtpub.com/support, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded on our website, or added to any list of existing errata, under the Errata section of that title. Any existing errata can be viewed by selecting your title from http://www.packtpub.com/support.

Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at <copyright@packtpub.com> with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

Questions

You can contact us at <questions@packtpub.com> if you are having a problem with any aspect of the book, and we will do our best to address it.

Chapter 1. Instant Dependency Management with RequireJS How-to

Welcome to *Instant Dependency Management with RequireJS How-to*. Here, we will learn the concepts of the **AMD** (**Asynchronous Module Definition**) specification and how using Require.js, an AMD script loader built by James Burke, can improve the performance and maintainability of a web application. Since Require.js can be used with other libraries, we will also learn how to integrate popular libraries such as jQuery, the jQueryUI Widget Factory, Backbone.js, Lodash.js, Jasmine, and more into an AMD workflow. We will also review popular Require.js plugins and see how to incorporate them to support common web development tasks including dynamically rendering JavaScript templates, optimizing JavaScript files (that is, concatenating, minifying), and more. Let the journey begin!

Using RequireJS (Intermediate)

Learn how to synchronously and asynchronously include Require.js within your HTML page to kick start your JavaScript application.

How to do it...

Use the following steps to include Require.js synchronously:

1. Include the Require.js library as an external JavaScript file within the head or body tag of the page.
2. Set the `data-main` attribute of the Require.js script tag `include` to the relative file location (relative to the HTML file) of the first JavaScript file that you would like Require.js to load.

> **Tip**
>
> The JavaScript file being specified within the `data-main` attribute does not need to include the `.js` file extension, as Require.js expects a JavaScript file.

3. The following code will synchronously include Require.js within your HTML page.

```
<!DOCTYPE html>
    <html>
```

```
    <head>
    <title>My Sample Project</title>
    <!-- data-main attribute tells
require.js to load
    js/main.js after require.js loads. -->
    <script data-main="js/main"
src="js/libs/require.js">
    </script>
    </head>
    <body>
        <h1 title="Sample Project">My
Sample Project</h1>
    </body>
</html>
```

Tip

Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at http://www.PacktPub.com. If you purchased this book elsewhere, you can visit http://www.PacktPub.com/support and register to have the files e-mailed directly to you.

To include Require.js asynchronously, append an HTML script tag, with the correct `src` and `data-main` attributes, to the page.

```
<!DOCTYPE html>
<html>
  <head>
    <title>My Sample Project</title>
    <script>
      // Script to include Require.js
        (function(w, d, undefined) {
          var script = d.createElement("script");
          script.setAttribute("data-main",
"js/main");
          script.src = "scripts/require.js";
          script.async = true;

 d.getElementsByTagName("head")
[0].appendChild(script);
        }(window, window.document));
    </script>
  </head>
  <body>
    <h1 title="Sample Project">My Sample
Project</h1>
  </body>
```

```
</html>
```

How it works...

In both of the previous code snippets, Require.js looks for a `main.js` file inside of a `js` folder. The JavaScript file, `main.js`, is the first JavaScript file loaded by Require.js, and contains all of our application's Require.js configurations.

The Require.js library is being included via the `src` attribute, using the relative HTML file path.

The script tag that is used to include Require.js uses the HTML5 data attribute, `data-main`, to tell Require.js that `main.js` is the first JavaScript file that it should load. Since `main.js` will be the first JavaScript file loaded by Require.js, it will set all application-level Require.js configurations.

Including Require.js on the page synchronously is an accepted practice, but you may choose to asynchronously include Require.js to achieve even better page load rendering performance.

Setting RequireJS configurations (Intermediate)

Require.js configurations allow you to set application-level AMD settings such as file alias names, relative file paths, and dependency management ordering.

Before we start to dynamically load files with Require.js, let's review how to use the global `require()` method to set Require.js configurations for our application.

How to do it...

All of our Require.js configurations should be stored in the first JavaScript file loaded by Require.js. To set Require.js configurations, you can pass a configuration object as a parameter to the `require()` method. These configurations will be used throughout the rest of our application:

```
// Configuration file
// ------------------
require.config({

    // Sets the js folder as the base directory
for all futurerelative paths
    baseUrl: "./js",

    // 3rd party script alias names
    paths: {

        // Core Libraries
        // --------------

        // jQuery
        "jquery": "libs/jquery",

        // Plugins
        // -------

        // Twitter Bootstrap jQuery plugins
        "bootstrap": "libs/plugins/bootstrap",

        // Application Files
        // -----------------

        // app.js
        "app": "app/app"

    },
```

```
    // Sets the configuration for your scripts
that are notAMD compatible
    shim: {

        // Twitter Bootstrap jQuery plugins
depend on jQuery
        "bootstrap": ["jquery"]

    }
});
```

How it works...

Let's go through the configuration options that were used in the previous code snippet:

- `baseUrl`: The beginning file path that is used for all of the file lookups with Require.js.
- `paths`: An object containing all of the mappings of file alias names to file paths. The `paths` configuration allows you to reference an alias name (that is jQuery), which can point to a folder/path or a single file. Referencing alias names is preferred over having to type the entire relative file path of a resource.
- `shim`: An object that manages dependencies (that is, a jQuery plugin depends on jQuery) and non-AMD compatible scripts. Examples of popular non-AMD compatible third-party scripts include Underscore.js and Backbone.js.

There's more...

This recipe demonstrates the most commonly used Require.js configurations, but is not an exhaustive list of all of the possible configurations. If you would like to look at the official Require.js configuration API, visit http://requirejs.org/docs/api.html#config.

Creating AMD modules (Advanced)

Before a JavaScript file can be loaded with Require.js, it must first register itself as an AMD module. If a file does not adhere to the AMD module specification, Require.js will not know how to trace any of its required file dependencies, which may result in a runtime error.

The AMD module specification requires a file to wrap itself within a global `define()` method and pass all of its file dependencies.

Both the `require()` and `define()` methods are global Require.js methods that will continue to play key roles when dynamically loading and registering files as AMD modules:

- `require()`: This is a method that is used to set Require.js configurations and dynamically load AMD modules
- `define()`: This is a method that wraps and registers individual files as AMD modules that can be loaded by the Require.js `require()` method

An AMD module can return any type of value (that is object, function, string) and defaults to `undefined` if no return value is specified. This recipe will cover how to create and reuse AMD modules.

How to do it...

Let's create an AMD module that does not specify a return value:

```
// AMD Module that returns undefined
// ---------------------------------

// The define method is passed an array of file
dependencies and a callback function
define(["jquery", "bootstrap"], function($) {

    // This is the same thing as
$(document).ready()
    $(function() {

        // Finds all buttons on the page and
calls the Twitter Bootstrap popover method
        $("button").popover();
    });
});
```

Let's next see how we can create an AMD module that returns an object literal:

```
// Simple name/value pair module
// ------------------------------

// The define method is passed a JavaScript
object literal
define({
    appName: "Example",
    production: true

});
```

The previous object literal AMD module example could also be rewritten using a function as follows:

```
// Object Module
// -------------

// The define method is passed a JavaScript
anonymous function
define(function () {

    // Returns an object literal
    return {
        appName: "Example",
        production: true
    }
});
```

Once you have created an AMD module, you can include it as a dependency to a different AMD module as follows:

```
// Dynamically loads an AMD module, app.js,
before the current module executes
define(["app"], function(Application) {

    // The current module callback function is
executedafter app.js is loaded
});
```

How it works...

Our first code example passes jQuery and the Twitter Bootstrap jQuery plugins as dependencies, and then calls the Twitter Bootstrap pop-over plugin on one of our DOM elements. Since this action will only be completed once in our application, it is not necessary to return any details about the module.

Tip

This example assumes that jQuery was listed as a dependency to the Twitter Bootstrap jQuery plugin module using the Require.js shim configuration.

The second AMD module example demonstrates a module that has no dependencies and returns an object literal. This type of AMD module is called a Simple **Name/Value Pair Module** and is a commonly-used format for configuration files. When looking at this example, you might be confused as to how Require.js knows to return anything since there is no explicit return statement being specified. In this case, the Require.js `define()` method is smart enough to know that the only parameter is an object literal and makes sure the object is returned internally.

Tip

It is important to note that The Simple Name/Value Pair Module only works with object literals. Strings are interpreted as module IDs, arrays are treated as file dependencies, and functions act as callbacks that execute module code once all dependencies have been loaded. This feature may not work on other AMD loaders.

The next AMD module example also returns an object literal, but uses a function format. This module is referred to as a **Simple Definition Function Module**. You may prefer to use this type of module since it is more obvious what is getting returned than the Simple Name/Value Pair Module.

Hopefully you are starting to see how the AMD module pattern promotes smaller, more reusable files. When your application starts to grow, this technique is critical for code maintainability.

An important concept to understand about Require.js modules is that returning a value (that is object, function, string, and so on) promotes reusability and local scoping, since other modules can now include the module as a dependency and access the returned value (despite a potential local scope).

There's more...

This recipe demonstrates the most commonly used Require.js AMD module definitions, but is not an exhaustive list of all of the possible AMD definitions. If you would like to look at the official Require.js module definition API, visit http://requirejs.org/docs/api.html#define.

Require.js also allows modules to be declared using the **Simplified CommonJS** wrapper syntax. If you are interested in the CommonJS format for declaring your modules, visit https://github.com/jrburke/requirejs/wiki/Differences-between-the-simplified-CommonJS-wrapper-and-standard-AMD-define#wiki-cjs for more information.

Using jQuery and jQueryUI Widget Factory plugins with RequireJS (Advanced)

This section will demonstrate how to use jQuery and jQueryUI Widget Factory plugins with Require.js. In case you are not familiar, the jQueryUI Widget Factory provides a consistent API for building jQuery plugins and has become a popular tool for jQuery plugin authors.

Since more and more jQuery plugins use the jQueryUI Widget Factory, it is necessary to understand how to use these jQuery plugins with AMD loaders, such as Require.js.

How to do it...

We must declare the `jquery` alias name within our Require.js configuration file.

```
require.config({
    // 3rd party script alias names
    paths: {
    // Core Libraries
     // --------------
      // jQuery
      "jquery": "libs/jquery",
    // Plugins
    // -------
    "somePlugin": "libs/plugins/somePlugin"
  }
});
```

If a jQuery plugin does not register itself as AMD compatible, we must also create a Require.js `shim` configuration to make sure Require.js loads jQuery before the jQuery plugin.

```
shim: {
    // Twitter Bootstrap plugins depend on jQuery
    "bootstrap": ["jquery"]
 }
```

We will now be able to dynamically load a jQuery plugin with the `require()` method.

```
// Dynamically loads a jQuery plugin using the
require() method
```

```
require(["somePlugin"], function() {

    // The callback function is executed after
the pluginis loaded
});
```

We will also be able to list a jQuery plugin as a dependency to another module.

```
// Sample file
// -----------

// The define method is passed a dependency array
and a callback function
define(["jquery", "somePlugin"], function ($) {
    // Wraps all logic inside of a jQuery.ready
event
    $(function() {
    });
});
```

When using a jQueryUI Widget Factory plugin, we create Require.js path names for both the jQueryUI Widget Factory and the jQueryUI Widget Factory plugin:

```
"jqueryui": "libs/jqueryui",

"selectBoxIt": "libs/plugins/selectBoxIt"
```

Next, create a `shim` configuration property:

```
// The jQueryUI Widget Factory depends on jQuery
"jqueryui": ["jquery"],

// The selectBoxIt plugin depends on both jQuery
and the jQueryUI Widget Factory
"selectBoxIt": ["jqueryui"]
```

We will now be able to dynamically load the jQueryUI Widget Factory plugin with the `require()` method:

```
// Dynamically loads the jQueryUI Widget Factory
plugin, selectBoxIt, using the Require() method
require(["selectBoxIt"], function() {

    // The callback function is executed after
selectBoxIt.js (and all of its dependencies) have
been loaded
});
```

We will also be able to list the jQueryUI Widget Factory plugin as a dependency to another module:

```
// Sample file
// -----------

// The define method is passed a dependency array
and a callback function
define(["jquery", "selectBoxIt"], function ($) {

    // Wraps all logic inside of a jQuery.ready
event
    $(function() {

    });
});
```

How it works...

Luckily for us, jQuery adheres to the AMD specification and registers itself as a named AMD module. If you are confused about how/why they are doing that, let's take a look at the jQuery source:

```
// Expose jQuery as an AMD module
if ( typeof define === "function" && define.amd
&& define.amd.jQuery ) {
    define( "jquery", [], function () { return
jQuery; } );
}
```

jQuery first checks to make sure there is a global `define()` function available on the page. Next, jQuery checks if the define function has an `amd` property, which all AMD loaders that adhere to the AMD API should have.

> **Tip**
>
> Remember that in JavaScript, functions are first class objects, and can contain properties.

Finally, jQuery checks to see if the `amd` property contains a `jQuery` property, which should only be there for AMD loaders that understand the issues with loading multiple versions of jQuery in a page that all might call the `define()` function.

Essentially, jQuery is checking that an AMD script loader is on the page, and then registering itself as a named AMD module (jquery).

Tip

Since jQuery exports itself as the named AMD module, `jquery`, you must use this exact name when setting the path configuration to your own version of jQuery, or Require.js will throw an error.

If a jQuery plugin registers itself as an anonymous AMD module and jQuery is also listed with the proper lowercased `jquery` alias name within your Require.js configuration file, using the plugin with the `require()` and `define()` methods will work as you expect.

Unfortunately, most jQuery plugins are not AMD compatible, and do not wrap themselves in an optional `define()` method and list `jquery` as a dependency. To get around this issue, we can use the Require.js `shim` object configuration like we have seen before to tell Require.js that a file depends on jQuery. The `shim` configuration is a great solution for jQuery plugins that do not register themselves as AMD modules.

Unfortunately, unlike jQuery, the jQueryUI does not currently register itself as a named AMD module, which means that plugin authors that use the jQueryUI Widget Factory cannot provide AMD compatibility. Since the jQueryUI Widget Factory is not AMD compatible, we must use a workaround involving the `paths` and `shim` configuration objects to properly define the plugin as an AMD module.

There's more...

You will most likely always register your own files as anonymous AMD modules, but jQuery is a special case. Registering itself as a named AMD module allows other third-party libraries that depend on jQuery, such as jQuery plugins, to become AMD compatible by calling the `define()` method themselves and using the community agreed upon module name, `jquery`, to list jQuery as a dependency.

Using Backbone.js and Lodash.js with RequireJS (Intermediate)

Backbone.js is a great client-side MV* JavaScript framework that provides structure to JavaScript applications by providing View, Model, Collection, and Router classes. Backbone also provides a publish/subscribe (pub sub) mechanism, by allowing each of its objects to trigger and listen to events. Although Backbone.js and Require.js perfectly complement each other, Backbone.js does not register itself as an AMD module.

Lodash.js is a utility library that provides both functional and helper methods without extending any of the native JavaScript objects and is a required dependency when using Backbone.js.

> ### Tip
>
> This recipe focuses on using Lodash.js instead of Underscore.js, since Lodash provides cross-browser bug fixes, a more consistent API, and better performance than Underscore.

This recipe describes how to register Backbone.js as an anonymous AMD module using the Require.js `shim` configuration.

How to do it...

Since Backbone.js does not register itself as an AMD module we must use the `shim` configuration to make Backbone AMD compatible.

```
// Require.js Configuration File
// -----------------------------------
require.config({

  // Sets the js folder as the base directory
  baseUrl: "./js",

  // The paths configuration can be used to
rename modules
  paths: {

      // Core Libraries
      // --------------
      "jquery": "libs/jquery",
      "underscore": "libs/lodash",
      "backbone": "libs/backbone"
 },
```

```
  // Registers Backbone as an AMD module
  shim: {

    // Backbone
    "backbone": {

      // Underscore/lodash and jQuery will be
loaded beforeBackbone
      "deps": ["underscore", "jquery"],

      // Uses the window.Backbone object as the
module value
       "exports": "Backbone"

     }

  }

});
```

We can next use Backbone.js within a Require.js module:

```
// Sample file
// -----------

// Registers an AMD module
define(["jquery", "backbone"], function ($,
Backbone) {

  // jQuery and Backbone are loaded before the
callback is executed

});
```

How it works...

The `deps` property, within the `shim` configuration, is an array of file alias names dependencies and the `exports` property is the global property that the script uses as the module value. The preceding code snippet is a great example of how the `shim` configuration can be used to make third-party scripts, which export a single global object, AMD compatible.

Once Backbone.js is AMD compatible, we are now free to use it within all of our Require.js modules.

Inside the Require.js modules, the jQuery and Backbone objects are passed to the top-level module callback function and can be used within the module body. The callback function parameters map directly to the array of dependencies that we passed as the first argument. The order is very important!

There's more...

Throughout the remainder of the book, we will be starting to use code snippets from my popular open source project, **Backbone-Require-Boilerplate**, to demonstrate even more techniques when using both Backbone.js and Require.js.

Backbone-Require-Boilerplate promotes decoupling your JavaScript into modules using Require.js, separating business logic from application logic using Backbone.js Collections, Models, and Views, reusing your JavaScript between desktop and mobile web versions while using a mobile framework (**jQuery Mobile**), including non-AMD compatible scripts in your project, optimizing all of your JavaScript files for production, and unit testing your JavaScript with the popular third-party framework, **Jasmine**.

Separating mobile web and desktop logic with RequireJS (Intermediate)

Developing a web application that works well for mobile web and desktop browsers is a visual and technical challenge. More and more developers are beginning to use both a mobile-specific framework (that is, jQuery Mobile) and a desktop-specific framework (that is, Twitter Bootstrap) to support as many devices as possible.

Because of performance concerns on mobile devices, it is important for applications to only include the necessary assets/resources for each device. Unfortunately, most developers do not know how to reuse and separate logic between mobile web and desktop versions of an application, so they create two completely separate applications.

The inability to adhere to the **DRY** (**Don't Repeat Yourself**) development principle results in developers maintaining two distinct codebases and being 50 percent less productive. If a change is made in the desktop codebase, the mobile web codebase must also be updated to be kept in sync. This cat/mouse development strategy is tiresome for developers and is not maintainable (especially for a small team).

This recipe describes how you can use the popular open source project, Backbone-Require-Boilerplate, to separate and reuse logic between an application's mobile web and desktop versions of an application. It is important to find a solution that allows our application to conditionally include different JavaScript and CSS files depending on which device and which environment mode (production or development) is being used.

How to do it...

Backbone-Require-Boilerplate provides a custom `loadFiles()` method that allows you to pass which CSS and JavaScript files you would like to conditionally load:

```
// Mobile/Desktop Detection script
(function(ua, w, d, undefined) {

  // App Environment
  // ---------------
  //  Tip: Set to true to turn on
"production" mode
  var production = true,
    filesToLoad;
```

```
          // Mobile/Tablet Logic

 if((/iPhone|iPod|iPad|Android|BlackBerry|
Opera Mini|IEMobile/).test(ua)) {

            // Mobile/Tablet CSS and JavaScript
files to load
            filesToLoad = {
              // CSS file that is loaded when in
development mode
              "dev-css": "css/mobile.css",
              // CSS file that is loaded when in
production mode
              "prod-css": "css/mobile.min.css",
              // Require.js configuration file
that is loaded whenin development mode
              "dev-js": { "data-main":
"js/app/config/config.js","src":
"js/libs/require.js" },
              // JavaScript initialization file
that is alsoloaded when in development mode
              "dev-init":
"js/app/init/MobileInit.js",
              // JavaScript file that is loaded
when in productionmode
              "prod-js":
"js/app/init/MobileInit.min.js",
            };

          }

          // Desktop Logic
          else {

            // Desktop CSS and JavaScript files
to load
            filesToLoad = {
              // CSS file that is loaded when in
development mode
              "dev-css": "css/desktop.css",
              // CSS file that is loaded when in
production mode
              "prod-css": "css/desktop.min.css",
              // Require.js configuration file
that is also loadedwhen in development mode
              "dev-js": { "data-main":
"js/app/config/config.js","src":
"js/libs/require.js" },
              // JavaScript initialization file
that is loadedwhen in development mode
              "dev-init":
"js/app/init/DesktopInit.js",
              // JavaScript file that is loaded
when in productionmode
```

```
                "prod-js":
"js/app/init/DesktopInit.min.js"
              };

            }

            boilerplateMVC.loadFiles(production,
filesToLoad,function() {
              if(!production && window.require) {
                require([filesToLoad["dev-init"]]);
              }
            });

        })(navigator.userAgent || navigator.vendor
|| window.opera,window, document);
```

How it works...

Looking at the preceding code example, we can see that all of the JavaScript is wrapped inside an **Immediately Invoked Function Expression** (**IIFE**). Everything inside of the IIFE is executed immediately and is locally scoped, which prevents the pollution of the global namespace. Although not shown in the code example, Backbone-Require-Boilerplate also places the `boilerplateMVC.loadFiles()` method inside the IIFE.

Inside the IIFE, the local variable, `production`, is set to `false` to tell the `loadFiles()` method that our application is in the development mode (this makes sure that the un-minified CSS and JavaScript files are included, which are easier to debug). If we would like to release our application to the outside world, we can set the production variable to `true`, to make sure that only the minified JavaScript and CSS files are included for the best performance.

Next, we use a **user agent** (**UA**) detection script to determine if a mobile, tablet, or desktop browser is being used. Depending on which type of browser is being used, we populate a JavaScript configuration object with two different arrays.

> ## Tip
>
> Although a UA detection script is used in the previous example, it may make more sense (depending on the scenario) to use feature detection to detect for mobile and tablet browsers.

These two arrays are used to tell the `loadFiles()` method which CSS and JavaScript files should be loaded for both development and production modes.

Once our Require.js configuration file is included on the page, we can use the magic of Require.js and AMD script loading to conditionally reuse and load any additional JavaScript files that we need.

There's more...

This recipe has demonstrated a great application structure for conditionally loading page assets, but has not gone in-depth on how to reuse your code. For a more complete example of code reuse, please check out the full Backbone-Require-Boilerplate project source on Github.

Using JavaScript templates with the RequireJS text plugin (Intermediate)

If you are a JavaScript developer, you have most likely embedded HTML within your JavaScript code using string concatenation. Many projects have tried to abstract this common task and implement a cleaner approach with JavaScript templates.

A common technique, when using a JavaScript template library, is to place a JavaScript template inside an inline script tag, with a content-type of text/template. Since the inline script tag has an unknown content-type, browsers will ignore it and not execute the script. This is a well-proven solution for including a small number of JavaScript templates, but soon becomes unmanageable for a large application. This recipe will demonstrate how the Require.js text plugin can be used with Lodash JavaScript templates for a more manageable solution.

How to do it...

The Require.js text plugin allows us to store JavaScript templates inside separate HTML files instead of embedding inline script tags, as follows:

```
   <!-- example.html -->
  <h1>Your Template says:</h1>
   // It is assumed that the data property has
been passed toour template
  <% if(data) { %>
     <h3 class='example'>Data has been passed to
ourtemplate!</h3>
   <% } else { %>
    <h3 class='example'>No data has been passed
to ourtemplate</h3>
  <% } %>
```

To use the Require.js text plugin to load the previous template example, we should first add a path alias name within our Require.js configuration settings:

```
// Sample Configuration file
// -------------------------
require.config({

  // Path alias names
  paths: {
```

```
        // Require.js Text Plugin
        "text": "libs/plugins/text"
    }

});
```

> **Tip**
>
> Setting a path name is not required, but it makes it easier for us to reference and use the text plugin inside our other modules.

Once we have set a path alias name for the text plugin, we can start including our lodash templates within our AMD modules using the `text!` prefix:

```
// Example
// -------
define(["jquery", "text!templates/example.html"],

    function($,temp){

        // Stores the compiled Lodash template
        var exampleTemplate = _.template(temp,
{ data: true });

        // Dynamically updates the UI with the
template
        $("#example").html(exampleTemplate);

    }
);
```

How it works...

Behind the scenes, the Require.js text plugin retrieves the contents of our JavaScript template (`example.html`) with an Ajax request, and passes the template content back to our module callback function.

> **Note**
>
> When using the Require.js optimizer for production (which we will look at in a later section of the book), all templates are inserted inline, saving an HTTP request.

Since the Require.js text plugin uses the `XMLHttpRequest` (**XHR**) object, in development mode, to fetch the text for the resources it handles, there are some restrictions due to the browser/web security policies. One of the most

important restrictions of the XHR object is that many browsers do not allow `file://` access to any file. This means that you should be running a local web server, when using the Require.js text plugin, or you may encounter a JavaScript cross-domain error.

In the earlier example, we prepended the word `text!` to our templates/`example.html` dependency, which tells Require.js to use the Require.js text plugin to dynamically load our JavaScript template.

Once our template is passed to our module callback function (via the `temp` parameter), we pass it to the Lodash `template()` method to make sure it is properly compiled to plain HTML (and stripped of any funny template syntax). After the template has been compiled into a string of HTML, we can dynamically add the template HTML content to the page using the jQuery `html()` method.

> **Tip**
>
> The jQuery `html()` method properly encodes an HTML string and updates the DOM content of the calling element.

There's more...

The text plugin is only one example of a Require.js loader plugin that allows you to load different types of resources as dependencies. Check out http://requirejs.org/docs/plugins.html to read more about other Require.js plugins and even how to use the Require.js API to write your own plugin.

Creating Jasmine unit tests with RequireJS (Intermediate)

Require.js improves the maintainability of an application, by promoting small and reusable modules, but it cannot prevent software defects. As the complexity of an application's codebase grows, it is common for new defects to be introduced and old defects to be reintroduced (regression bugs). Although no software application is 100 percent defect free, **JavaScript unit testing** can be used to minimize the number of possible new and recurring defects.

There are many open source JavaScript unit testing frameworks available to help developers test their JavaScript applications with Require.js. One popular framework is Jasmine, a **behavior-driven development** (**BDD**) framework. This recipe will demonstrate how to use Jasmine with Require.js to test an application's JavaScript code.

How to do it...

Before we start writing unit tests, we need to first create an HTML file where our Jasmine test results (showing which tests passed or failed) will be displayed:

```
<!doctype html>
<html>
<head>
  <title>Jasmine Spec Runner</title>
  <link rel="stylesheet" type="text/css"
href="css/jasmine.css">
  <script src="js/libs/jasmine.js"></script>
  <script src="js/libs/jasmine-html.js"></script>
  <script src="js/libs/require.js"
data-main="js/app/config/config"></script>
</head>
<body>
    <script>
      // Add your Jasmine spec files here
      require(["../../test/specs/spec"],
function(spec) {
        jasmine.getEnv().addReporter(new
jasmine.TrivialReporter());
        jasmine.getEnv().execute();
      });
    </script>
</body>
</html>
```

We will now be able to start writing our Jasmine unit tests:

```
// Jasmine Unit Testing Suite
// --------------------------
define(['app'], function(App) {
  describe("A test suite", function() {
    it("contains a spec (test) with an
expectation",function() {
      expect(App.test).toBe(true);
    });
  });
});
```

How it works...

Within our Jasmine test result's HTML page, we include the Jasmine CSS file (used to style our test results page), the Jasmine JavaScript files (`jasmine.js` and `jasmine-html.js`), and Require.js. The script tag that includes Require.js sets its `data-main` attribute to the relative file location of `config.js` (our Require.js configurations file), which sets our module's `paths` and `shim` configurations.

> **Tip**
>
> Including Jasmine in the page, before Require.js, ensures that the global Jasmine object is available.

After all of the Jasmine dependencies and Require.js configurations have been loaded, we use the `require()` method to dynamically load our Jasmine `spec` file, which includes all of our Jasmine unit tests.

> **Tip**
>
> Since this is a basic example, we only have one file with unit tests (`spec.js`), so only one path is set.

Once our Jasmine `spec` file is loaded, Jasmine begins to run all of our unit tests.

We have seen how to properly load Jasmine unit tests with Require.js, but it is now time to review and understand an actual unit test. We first wrap the entire file inside a `define()` method to be AMD compatible.

> **Tip**
>
> The `define` wrapper is only needed if you return a value or if the spec has any dependencies.

Next, we use the Jasmine `describe` statement to wrap one of our unit tests inside a suite. A **suite** in Jasmine is used to group similar unit tests together to help communicate what category each test falls under. Our example only includes one unit test, but it is common to have many unit tests inside each suite, and many different suites.

Inside our test suite, we are using the standard Jasmine unit test syntax for one of our unit tests. Jasmine uses the elegant *it and expect* syntax so that tests can be read and communicated in close to plain English. After writing many Jasmine unit tests, you will find that your tests are useful not only for testing, but as a low-level description of your application.

There's more...

This recipe provided a simplistic example of basic Jasmine testing with Require.js, but your application will most likely be more complex. If you are using jQuery, a great addition to your Jasmine tool belt is the `jasmine-jquery jQuery` plugin (https://github.com/velesin/jasmine-jquery).

The `jasmine-jquery` plugin provides a set of custom DOM matchers (that is, `expect($('<input type="checkbox" checked="checked"/>')).toBeChecked()`) that help make your Jasmine client-side tests more elegant. The plugin also provides a feature called HTML fixtures, which allows you to dynamically load different HTML content that is used in your tests. You can see the `jasmine-jquery` plugin in action (along with a more advanced Jasmine/Require.js example) in the Backbone-Require-Boilerplate project.

Using the RequireJS optimizer (Advanced)

An AMD project, built with Require.js, consists of advanced dependency management techniques to manage script execution and load order. Because of this unique management of script dependencies, it is impossible to optimize (minify/concatenate) an entire project with a normal third-party tool, such as *Google Closure Compiler* or *UglifyJS*, without an AMD-aware tool to bridge the gap. Luckily for us, *James Burke* (the creator of Require.js) has also created **r.js**, a command-line tool for optimizing an entire AMD/Require.js project.

This recipe will focus on setting up and running the Require.js optimizer for a mobile web/desktop project on the command line with **node.js**, a JavaScript command-line environment. We will also learn how to incorporate **almond.js**, a minimal/smaller AMD replacement loader for Require.js, to further minimize an application's footprint.

How to do it...

To use the Require.js optimizer, we need to create a build profile to hold all of our optimization settings:

```
// app.build.js

// Install node.js, navigate to the app.build.js
file path, and then run this command: "node r.js
-o app.build.js"
({

  // Creates a js-optimized folder and puts the
entire optimized project there
  dir: "../js-optimized",

  // Tells Require.js to look at config.js for
all shim and pathconfigurations
  mainConfigFile: "config.js",

  // Includes app.js and all of its dependencies
in the build
  modules: [

    {
      name: "app"
    },
  ]
```

```
})
```

If we want to use Almond.js instead of Require.js in production, we need to update our build profile as follows:

```
// app.build.js

// Install node.js, navigate to the app.build.js
file path, and then run this command: "node r.js
-o app.build.js"
({

// Tells Require.js to look at config.js for all
shim and path configurations
mainConfigFile: "config.js",

// The optimized build file will use almond.js
(AMD shim)
name: "libs/almond"

// Includes app.js and all of its dependencies in
the build
include: ["app"],

// The output app optimized file
out: "app.min.js"

// Wraps all scripts in an IIFE to prevent global
namespace pollution
wrap: true,

})
```

How it works...

The basic r.js build profile example takes a minimalist approach to optimizing an entire mobile web/desktop application. The first build optimization property we used is `dir`, which is the directory path used to save the output.

> **Tip**
>
> If we do not set the `dir` property, the optimized project will be saved in a `build` directory in the same directory as the `build` file.

Next, we use the `mainConfigFile` property to tell the optimizer where to look for all `shim` and `paths` configurations.

Finally, the `modules` property is set to tell the optimizer which project files to optimize. In our example, we are assuming that `app.js` is the entry point to our application, which means that all of our project files are referenced and optimized as dependencies.

To run the Require.js build, first make sure that `node.js` is installed, and then navigate to the directory holding `app.build.js` and type:

```
node r.js -o app.build.js
```

Once we have built our optimized files, we just have to update our Require.js script including the `data-main` property to reflect the new path to our application (the `js-optimized` folder).

Although our basic r.js example correctly uses the Require.js optimizer, we are still using Require.js to load our optimized files in production. If an application is optimizing all files into one for production builds and does not dynamically load any files, it may be preferable to use the AMD shim, Almond.js.

> **Tip**
>
> It is important to remember that if you use Almond.js in production, you are not allowed to dynamically load scripts. Only small to medium applications, that load only one file, should be using Almond.

Almond.js is much smaller than Require.js (around 1 KB when minified with Google Closure Compiler and gzipped), and supports the Require.js text plugin.

Within our second build profile, we use the `name` and `include` properties together to embed Almond.js within one of the optimized JavaScript files of our application (this limits the number of HTTP requests). We also set the `wrap` property to `true` so that all of our JavaScript code is locally scoped (including the `define()` and `require()` methods). Finally, we use the `out` property to tell Require.js where to output one of our minified files. This approach is different from our first example, which had Require.js create a brand new project folder.

There's more...

Look at the Backbone-Require-Boilerplate example at https://github.com/BoilerplateMVC/Backbone-Require-Boilerplate/blob/master/Gruntfile.js to see how you can use Grunt.js to create a Require.js optimizer build profile for both mobile web and desktop

versions of your application. **Grunt.js** is a JavaScript command-line task runner that allows you to easily automate common development tasks such as code linting, minification, and unit testing. It is an essential tool that makes it easy to automate common tasks.