Mastering PhoneGap Mobile Application Development

This book will guide you through the process of creating a complex data-driven hybrid mobile application using PhoneGap, web technologies, and third-party plugins.

You will first learn how to create a useful workflow to make development easier. From there, the next version of JavaScript (ES6) and the CSS pre-processor SASS are introduced to simplify creating the look of the mobile application. Responsive design techniques are also covered, including the flexbox layout module. As many apps are data-driven, you’ll build an application that relies upon IndexedDB and SQLite. You’ll also learn how to transfer files to and from external servers. When the app is complete, the book will guide you through the steps necessary to submit your app to the Google Play and Apple iTunes stores.

Who this book is written for
If you have created simple applications using PhoneGap in the past and now want to take your workflow and apps to the next level, this book will help you reach your goals. You should have a good working knowledge of HTML, CSS, and JavaScript, and prior experience with PhoneGap.

What you will learn from this book
- Construct build workflows that simplify complex application development
- Integrate the next version of JavaScript to simplify your code
- Create accessible hybrid applications
- Get to grips with various responsive design techniques
- Build an application that relies upon IndexedDB and SQLite
- Persist and query data using third-party database plugins
- Learn how to transfer files to and from external servers
- Create icons and splash screens suitable for submission to app stores
- Publish your app to the Google Play and Apple iTunes stores

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 1 'Task Automation'
- A synopsis of the book’s content
- More information on Mastering PhoneGap Mobile Application Development
**Kerri Shotts** has worked with computers for nearly 25 years. Her love for technology and programming started when she was introduced to her first computer: a Commodore 64. She obtained a degree in computer science while at college, and moved on to become a software test engineer. Afterward, she became an Oracle Database Administrator for several years. Now, she works as a technology consultant, creating, implementing, and maintaining custom applications (both desktop and mobile), websites, graphics and logos, and more for her clients. You can find her blog posts on her website ([http://www.photokandy.com/](http://www.photokandy.com/)) and she is active on the Google Groups for PhoneGap. When she isn’t working, she enjoys photography, music, and fish keeping. She is the author of several books published by Packt Publishing.
PhoneGap/Cordova, as a technology to create hybrid mobile apps, relies heavily upon JavaScript, HTML, and CSS in order to present your apps to your users. This is ideal in many ways, especially since you can rely upon your knowledge of web-based technologies in order to create cross-platform mobile apps. Being able to build on your existing knowledge set is a major plus when it comes to recommending Cordova, and generally, one can build their first simple app using Cordova pretty easily. But when it comes to building larger, more complex apps, it is useful to explore various technologies and tools that allow us to more efficiently develop mobile apps. The first few chapters of this book focus exactly on this need. We’ll cover task runners such as Gulp, packagers such as Browserify, and a method of writing CSS that’s easier to maintain using Sassy CSS.

In this book we generally refer to PhoneGap and Cordova simply as Cordova — PhoneGap itself is a distribution of Cordova and supplies additional features and utilities. If there is a specific difference we need to mention, we will do so in at the appropriate time.

One particular hallmark of large, complex apps is a requirement for a way to store complex data efficiently. There comes a point when using the Local Storage and File API become unwieldy, and so it is important to learn about other methods to store data, such as IndexedDB and Web SQL Database.

Quite often, it’s also necessary for apps to transfer large amounts of data between the device and external servers. You can imagine a social photography app might need to upload images, and an e-book app would need to download files that represent books. We’ll devote an entire chapter to this topic as well.
Of course, most developers want to get their apps out into the devices of as many
users as possible. This is why creating accessible applications is important. Not every
user has perfect vision. Some users may have trouble reading small text or text with
low contrast. Other users may have problems discerning various color shades. And
other users may not be able to hear your app's sounds very well. With assistive
technologies on many mobile platforms, it is a very good idea to build our apps so
that as many users as possible can use them effectively.

Deployment, ultimately, is our goal, and so the latter portion of the book will deal
exactly with that: how to create launch screens and icons for our app, how to create a
release build, and ultimately, how to deploy the app to the various app stores.

Along the way, we'll also deal with other concepts, such as tips you can use to make
your app’s user interface respond appropriately to the various form factors it finds
itself running on. You'll also find tools that can verify that your app is working as
you expect, as well as tips on how to find and fix performance issues.

When you're finished with this book, you should hopefully have the knowledge
necessary to tackle large scale and more complex apps that are accessible,
performant, and responsive.

In this book, we'll focus on the iOS and Android platforms. Cordova, however,
supports many other platforms. In general, most of what is covered in this book
applies to these other platforms as well. However, there are some third-party plugins
that are used. If you do want to support another platform, you'll want to verify that
the same or a similar plugin is available.

What this book covers

Chapter 1, Task Automation, introduces you to the process of automating your
common development tasks, including copying and transforming files in various
ways as part of your build steps. Sections deal specifically with creating an extensible
build system that can transpile JavaScript, minify code automatically, perform
Cordova CLI tasks, and lint your code to catch syntax errors.

Chapter 2, ECMAScript 2015 and Browserify, is a short introduction to many of the
new features in ECMAScript 2015 and beyond, including string interpolation, object
destructuring, named and default parameters, lexically bound functions, and more.
The chapter also introduces Browserify as a way to package your own code and
reuse great code modules from other JavaScript developers.

Chapter 3, Sassy CSS, introduces you to the world of CSS transpilers, notably the
Sassy CSS language. Specifically, the chapter covers variables, nesting, and mixins,
all of which make it easier to write readable and maintainable CSS code.
Chapter 4, More Responsive Design, focuses on the steps and features you can use to create hybrid apps that respond appropriately to the form factors of various devices. This chapter explores logical and physical pixels, important CSS units, media queries, image sizing, and using the flex box model to design complex yet responsive user interfaces.

Chapter 5, Hybrid Application Accessibility, explores the various methods you can use as a developer to make your app accessible to users who may need assistance seeing, hearing, or utilizing the content within your application.

Chapter 6, Testing and UI Automation, addresses the very real need to ensure that the apps we build actually function correctly. The chapter introduces you to testing concepts as well as various tools to automate tests. Finally, the chapter addresses how to automate the user interface as another method to test on real devices.

Chapter 7, IndexedDB, introduces you to a method of persistent storage other than Local Storage or the File API. The chapter covers how to create new object stores, save, retrieve and search for data, and more.

Chapter 8, Web SQL Database, introduces you to relational databases and how they can be used within Cordova using a third-party plugin. The chapter focuses on how to store and retrieve data using SQL.

Chapter 9, Transferring Files, covers how to download content from an external server into your app as well as how to upload content from your app and transfer it to an external server.

Chapter 10, Performance, discusses methods you can use to check how well your application performs on real devices, and also provides tips you can use to improve the performance if necessary.

Chapter 11, Graphical Assets, discusses how to create launch screens and icons for your application. The chapter provides tips on how to create a memorable icon and a good launch screen.

Chapter 12, Deployment, shows you how to create developer accounts for the Google Play Market and Apple App Store step by step so that you can deploy your apps to the world. Once your accounts are created, the chapter guides you through the process of uploading your app’s graphical assets, defining metadata, and finally, uploading your app itself.
Task Automation

While developing your app, there are often many tasks that need to be executed on a recurring basis. Although these tasks are rarely difficult or terribly time-consuming, over time, the effort adds up and it quickly becomes tiresome and error-prone.

Task automation simplifies these tiresome rituals. Automation lets you define the steps for tasks that you frequently execute (and even those that you execute infrequently, which may be even more useful). In a way, you could consider task automation similar to macros in other productivity applications you might use (such as Microsoft Word).

Individual tasks can also depend on other tasks, so you can simplify your manual processes to one or two easy-to-remember and easy-to-type commands. Furthermore, most task automation utilities provide a mechanism to watch for changes made to your project, automatically executing various tasks when any changes have been detected.

Technically, task automation isn't required while developing a PhoneGap / Cordova app; but as your apps grow larger and more complex, it becomes increasingly beneficial. There is some initial overhead, of course, which is often why small projects never implement task automation. But when an app has several views, lots of modules, and a good number of dependencies, the initial overhead quickly pays off.
Although this book is titled *Mastering PhoneGap Mobile Application Development*, we will be using *Cordova* to refer to PhoneGap and Cordova. PhoneGap is derived from Cordova and everything we do using Cordova will also work with PhoneGap. Where this doesn't hold true, we'll mention it explicitly.

Also, when we're referring to Cordova and PhoneGap, we are referring to the command-line utilities. There is a PhoneGap Build service available that performs compilation and packaging in the cloud; but if you want to use it, you'll need to adapt the content in this book appropriately. If you want to learn more, see the `README.md` file in the code package for this book.

There are several different task automation utilities available. Because one generally writes the majority of their Cordova app in HTML, CSS, and JavaScript, it makes sense to select a task automation system based on JavaScript. At the time of this writing, Gulp ([http://gulpjs.com](http://gulpjs.com)) and Grunt ([http://gruntjs.com](http://gruntjs.com)) are the most popular of the various available utilities.

In this chapter, you will learn about:

- Logology, the demonstration app
- Why use Gulp for Task Automation
- Setting up your app's directory structure
- Installing Gulp
- Creating your first Gulp configuration file
- Creating a modular Gulp configuration
- Copying assets
- Performing substitutions
- Executing various Cordova tasks
- Managing version numbers
- Supporting ES2015
- Linting your code
- Minifying/uglifying your code
Before we begin

This book comes with a code bundle that is available at https://github.com/kerrishotts/Mastering-PhoneGap-Code-Package. If you haven't downloaded it yet, I strongly advise you to do so. It contains all the code for each chapter as well as lots of snippets that demonstrate some of the examples in most chapters. Furthermore, the chapters in the book focus mostly on snippets—to see the topics in use in an actual application, you'll definitely want to look at the demonstration app's code.

Before continuing with this chapter, ensure that you have met the pre-requisites as listed in this book's preface. Software and hardware requirements are also listed in the code package for this book in the README.md file.

If you want to build and deploy the demonstration application from the code bundle, you'll need to install the earlier mentioned tools. Because the Cordova projects and platform-specific files are considered build artifacts, you'll need to execute the following in each chapter's directory in order to build each version of the app:

# On Linux / Mac OS X (using Bash shell)
$ npm install && gulp init

% On Windows
> npm install
> gulp init

About Logology

Before we go any further, let's describe the demonstration app we'll be building through the course of this book.

I've called it Logology. If you're familiar with any Greek words, you might have already guessed what the app will be: a dictionary. Now, I understand that this is not necessarily the most amazing app, but it is sufficient for our purposes. It will help you learn how advanced mobile development is done.

By the end of the book, the app will have the following features:

- **Search**: The user will be able to search for a term
- **Responsive design**: The app will size itself appropriately to any display size
- **Accessibility**: The app will be usable even if the user has visual difficulties
- **Persistent storage**: The app will persist settings and other user-generated information
Although the app sounds relatively simple, it's complex enough to benefit from task automation. Since it is useful to have task automation in place from the very beginning, in this chapter we'll install Gulp and verify that it is working with some simple files first. As such, the app in the code package for this first chapter is very simple; it exists solely to verify that our tasks are working correctly.

You may think that working through configuring task automation is very time-consuming, but it will pay off in the long run. Once you have a workflow that you like, you can take the workflow and apply it to any other apps you may build in the future. This means that future apps can be started almost immediately (just copy the configuration from the previous app). Even if you don't write other apps, the time you save from having a task runner will outweigh the initial setup time.

Why use Gulp for task automation?

Gulp ([http://gulpjs.com](http://gulpjs.com)) is a task automation utility using the Node.js platform. Unlike some other task runners, one configures Gulp by writing a JavaScript code. The configuration for Gulp is just like any other JavaScript file, which means that if you know JavaScript, you can start defining the automation tasks quickly.

Gulp also uses the concept of streams (again, from Node.js). Although you can think of a stream as a file, streams are actually more powerful. Plugins can be inserted within stream processing to perform many different transformations, including beautification or uglification, transpilation (for example, ECMAScript 6 to ECMAScript 2015), concatenation, packaging, and much more.

Gulp also tries to run as many dependent tasks in parallel as possible. Ideally, this makes it possible to run Gulp tasks faster, although this really depends on how your tasks are structured. Other task runners such as Grunt perform their task steps in a sequence that may result in a slower output, although it may be easier to follow the steps from input to output when they’re performed sequentially.

That's not to say that Gulp is the best task runner—there are many that are quite good, and you may find that you prefer one of them over Gulp. The skills you will learn in this book can easily be transferred to other task automation utilities.
Here are some other task runners that are useful:

- **Grunt** ([http://www.gruntjs.com](http://www.gruntjs.com)): The configuration is specified through settings, not code. The tasks are performed sequentially.
- **Cake** ([http://coffeescript.org/documentation/docs/cake.html](http://coffeescript.org/documentation/docs/cake.html)): It uses CoffeeScript and the configuration is specified via code, as it is seen in Gulp. If you like using CoffeeScript, you might prefer this over Gulp.
- **Broccoli** ([https://github.com/broccolijs/broccoli](https://github.com/broccolijs/broccoli)): It also uses configuration through code.

### Setting up your app's directory structure

Before we install Gulp, we should create the directory structure for our app. Keep in mind that there's no single correct way to structure your application, and your opinion on how apps should be structured is likely to change as you gain more experience. That said, this section will show you how I like to structure my projects.

My typical structure starts with the project's root directory. If you look at the code bundle for this book, you'll notice that the project's root directory is called `logology-v01/`.

I wouldn't normally append the version number on a project—that's what a version control system is for. However, since it is important that you be able to see changes from version to version, the code package splits these changes out by chapter—hence the version number.

Within the project's root directory are some additional directories:

- **config/**: Configuration files needed during the tasks are stored in this directory.
- **src/**: All the app's source code and image assets are stored in this directory. This is the source that we supply to Gulp. Gulp then transforms the source and stores it in a directory of our choosing (typically the `build` directory).
- **build/**: This directory contains the transformed HTML, CSS, and JavaScript code, as well as the native portions of a Cordova project.

The `build/` directory will not be present in the code bundle for this book. It is considered a build artifact, and as such, you can always regenerate it.
Within the `src/` directory lives our app's source code. I like to structure the code and assets as follows:

```bash
project-root/
src/
  config.xml            # Template file for our Cordova app's configuration
  res/                  # Icons & splash screens (covered in Chapter 11)
  www/                  # HTML, JavaScript, and CSS, and other web assets
  index.html          # Initial HTML file (as specified in config.xml)
  html/               # Additional HTML files, if any
  img/                # Image files, if we need them
  scss/               # Sassy CSS files (see Chapter 3)
  lib/                # Utility functions
  themes/           # Themes (appearance of the app)
  views/            # Styles specific to views in our app
  js/                # JavaScript
   lib/            # Third-party library code and support code
  app/            # Our application code
    index.js        # The entry point for our app
    controllers/    # View controllers live here
    lib/           # App-specific utility files
    localization/  # Language translations
    models/        # Data models go here
    views/         # Views and templates live here
```

If you look at the directory structure of this chapter in the code bundle, you will notice that a lot of it is missing. This is because it's not necessary at this point; we'll fill it out in the future chapters.
If you're wondering where the Cordova files are, you're paying attention. There aren't any. Yet. This is because they are considered to be build artifacts. Build artifacts are files that are created when you compile your app. If this feels both a little strange and a little familiar at the same time, there's a good reason behind it: the Cordova projects already have portions that are considered to be build artifacts. The strange part is that you're probably used to editing the www/ folder within the Cordova project, and executing cordova build to create the remaining build artifacts (namely, the native wrappers around your code, typically in platforms/).

In this book, however, we're going to take a higher level approach and consider the entire Cordova project as a build artifact. Why? Because Cordova has been extended by several other projects (such as Steroids: http://www.appgyver.com/steroids) and they usually have their own project formats and build steps. If you ever want to target these platforms, you can readily do so since your code doesn't live within a Cordova project. Furthermore, you might find that you want to target other technologies entirely, such as Electron (http://electron.atom.io) which encapsulates your code with a Chromium webview suitable for desktop execution. The build steps and project structure for Electron are different than what you might expect for a Cordova project. In short, it's a way to avoid tying yourself down.

All said, when we're done with the chapter, you'll have a Cordova app filled with your source code. That project will be present in the build/ directory.

If you ever need to execute Cordova commands outside Gulp, you'll need to change to the build/ directory first or the command will fail. This is because the Cordova CLI expects be to run within a Cordova project, and our app's root directory isn't a Cordova project. Only build/ contains a valid Cordova project.

A crucial part of our workflow is going to be our project's package.json file. This file will contain the app's version information, Cordova configuration, and more. If you're starting from scratch, you will need to create this file yourself by changing to the project's root directory and executing npm init:

If you are using the code bundle for this chapter, the package.json file is already built for you.

# (in your project's root directory)
$ npm init [ENTER]
This utility will walk you through creating a package.json file. It only covers the most common items, and tries to guess sane defaults.
name: (logology-v01) Logology [ENTER]
version: (1.0.0) [ENTER]
description: Logology and PhoneGap demonstration app [ENTER]
entry point: (index.js) [ENTER]
test command: [ENTER]
git repository: [ENTER]
keywords: dictionary word study phonegapcordova html5 javascript
css [ENTER]
author: Kerri Shotts<kerrishotts@gmail.com> [ENTER]
license: (ISC) MIT [ENTER]
About to write to .../logology-v01/package.json:

Is this ok? (yes) [ENTER]

At this point, you have the package.json file is created, but it will need a few more edits. Open the package.json file in your favorite editor and remove the scripts section. Then, add the following (for the full contents of this file you can refer to the code package):

```
{
    "cordova": {
        "name": "Logology",
        "id": "com.packtpub.logologyv1",
        "description": "Dictionary application",
        "author": {
            "name": "Kerri Shotts",
            "email": "kerrishotts@gmail.com",
            "site": "http://www.photokandy.com"
        },
        "template": "./blank",
        "platforms": [ "ios", "android" ],
        "preferences": {
            "permissions": "none",
            "fullscreen": "false",
            "orientation": "default",
            "stay-in-webview": "false",
            "ShowSplashScreenSpinner": "false",
            "AutoHideSplashScreen": "false",
            "disable-cursor": "false",
```
"KeyboardDisplayRequiresUserAction": "false",
"target-device": "universal",
"prerendered-icon": "true",
"webviewbounce": "false",
"DisallowOverscroll": "true",
"exit-on-suspend": "false",
"deployment-target": "7.0",
"detect-data-types": "false",
"SupressesIncrementalRendering": "true",
"android-minSdkVersion": "14",
"android-installLocation": "auto",
"android-windowSoftInputMode": "adjustResize",
},
"plugins": [
  "cordova-plugin-device@1.1.0",
  "cordova-plugin-network-information@1.1.0",
  "cordova-plugin-globalization@1.0.2",
  "cordova-plugin-whitelist@1.1.0",
  "ionic-plugin-keyboard@1.0.8",
  "cordova-plugin-inappbrowser@1.0.1"
]
}
}

The preceding code should be fairly self-explanatory. With it, we are essentially duplicating the contents of Cordova's config.xml file. Because the Cordova project itself is considered to be a build artifact, it makes sense to manage plugins, platforms, and preferences somewhere else and, because package.json handles the other configuration aspects of our project, it makes sense to include these configuration settings here.

This doesn't remove the need for a config.xml file. We'll cover this later on in this chapter.

At this point, we're ready to install Gulp and any other dependencies our project might need.
Installing Gulp

Installing Gulp is easy, but is actually a two-step process. The first step is to install Gulp globally. This installs the command-line utility; but Gulp won't work without also being installed locally within our project. If you aren't familiar with Node.js, the packages can be installed locally and/or globally. A locally installed package is local to the project's root directory, while a globally installed package is specific to the developer's machine. Project dependencies are tracked in package.json, which makes it easy to replicate your development environment on another machine.

Assuming you have Node.js installed and package.json created in your project directory, the installation of Gulp will be very easy. Be sure you are positioned in your project's root directory and then execute the following:

$ npm install -g gulp@3.9.0
$ npm install --save-dev gulp@3.9.0

If you receive an error while running these commands on OS X, you may need to run them with sudo. For example: sudo install -g gulp.

You can usually ignore any WARN messages.

Notice that we're specifying version numbers here – these are the versions that I used while writing the code for this book. You can try later versions if you want, as long as they are minor revisions. Major revisions may work, but you may also have to make modifications to the code in this book in order to support them.

It's a good idea to be positioned in your project's root directory any time you execute an npm or gulp command. On Linux and OS X, these commands generally locate the project's root directory automatically; but this isn't guaranteed on all platforms, so it's better to be safe than sorry.

That's it! Gulp itself is very easy to install, but most workflows require additional plugins that work with Gulp. In addition, we'll also install the Cordova dependencies for this project.

If you're working with the code bundle for this chapter, you can install all the following dependencies by executing npm install.
First, let's install the Cordova dependencies:

```bash
$ npm install --save-dev cordova-lib@5.4.1 cordova-ios@3.9.2 cordova-android@4.1.1
```

cordova-lib allows us to programmatically interact with Cordova. We can create projects, build them, and emulate them—everything we do with the Cordova command line can be done with cordova-lib. cordova-ios and cordova-android refer to the iOS and Android platforms that cordova platform add ios android would add. We've made them dependencies for our project, so we can easily control which version we will build it with.

While starting a new project, it's wise to start with the most recent version of Cordova and the requisite platforms. Once you begin, it's usually a good practice to stick with a specific platform version, unless there are serious bugs or security issues that require updating to a newer platform version.

Next, let's install the Gulp plugins we'll need:

```bash
$ npm install --save-dev babel-eslint@4.1.1 cordova-tasks@0.2.0 gulp-babel@5.2.1 gulp-bump@0.3.1 gulp-concat@2.6.0 gulp-eslint@1.0.0 gulp-jscs@3.0.2 gulp-notify@2.2.0 gulp-rename@1.2.0 gulp-replace-task@0.10.1 gulp-sourcemaps@1.3.0 gulp-uglify@1.4.0 gulp-util@3.0.6 merge-stream@1.0.0 require-all@1.1.0 rimraf@2.4.3
```

These will take a few moments to install; but when you're done, take a look at package.json. Notice that all the dependencies we added are also added to the devDependencies section in the file. This makes it easy to install all the project's dependencies at a later date (say, on a new machine) simply by executing npm install.

Before we go on, let's quickly go over what each of the earlier mentioned utilities do. We'll go over them in more detail as we progress through the remainder of this chapter:

- **gulp-babel**: Converts ES2015 JavaScript into ES5. If you aren't familiar with ES2015, it has several new features and an improved syntax that make writing mobile apps easier. Unfortunately, because most browsers don't yet natively support the ES2015 features and syntax, it must be transpiled to the ES5 syntax. Of course, if you prefer other languages that can be compiled to ES5 JavaScript, you could use those as well (these would include CoffeeScript and so on).
- **gulp-bump**: This small utility manages the version numbers in package.json.
- **gulp-concat**: This concatenates streams together. We can use this to bundle files together.
**Task Automation**

- **gulp-jscs**: This performs the JavaScript code style checks against your code. It supports ES2015.

- **gulp-eslint**: This lints your JavaScript code. It supports ES2015.

- **babel-eslint**: This provides ES2015 support to gulp-eslint.

- **gulp-notify**: This is an optional plugin, but it comes in handy, especially when some of your tasks take a few seconds to run. This plugin will send a notification to your computer's notification panel when something of import occurs—perhaps an error or a completion message. If the plugin can't send it to your notification panel, it will be logged to the console.

- **gulp-rename**: This renames the streams.

- **gulp-replace-task**: This performs text searches and replaces within the streams.

- **gulp-sourcemaps**: While transpiling ES2015 to ES5, it can be helpful to have a mapping between the original source and the transpiled source. This plugin creates them automatically for you.

- **gulp-uglify**: This uglifies/minifies your code. While useful for code obfuscation, it also reduces the size of the code.

- **gulp-util**: This provides additional utilities for Gulp, such as logging.

- **merge-stream**: This merges multiple tasks.

- **require-all**: This lets us import an entire directory of code into an object at once.

- **rimraf**: Easy file deletion. Akin to `rm` on the command line.

---

**Creating your first Gulp configuration file**

Gulp tasks are defined by the contents of the project's `gulpfile.js` file. This is a JavaScript program, so the same skills you have with JavaScript will apply here. Furthermore, it's executed by Node.js, so if you have any Node.js knowledge, you can use it to your advantage.

![Lightbulb Icon]

This file should be placed in the root directory of your project and must be named `gulpfile.js`. 
The first few lines of your Gulp configuration file will require the Gulp plugins you’ll need to complete the tasks. The following lines will then specify how the various tasks need to be performed. For example, a very simple configuration might look as follows:

```javascript
var gulp = require("gulp");
gulp.task("copy-files", function () {
  gulp.src(["./src/**/*"])
    .pipe(gulp.dest("./build"));
});
```

This configuration only performs one task: it moves all the files contained within `src/` to `build/`. In many ways, this is the simplest form of a build workflow, but it’s a bit too simple for our purposes.

Note the pattern we used to match all the files. If you need to see the documentation on what patterns are supported, see https://www.npmjs.com/package/glob.

To execute the task, you can execute `gulp copy-files`. Gulp would then execute the task and copy all the files from `src/` to `build/`.

What makes Gulp so powerful is the concept of task composition. Tasks can depend on any number of other tasks and those tasks can depend on yet more tasks. This makes it easy to create complex workflows out of simpler pieces. Furthermore, each task is asynchronous, so it is possible for many tasks with no shared dependencies to operate in parallel.

Each task, as you can see in the prior code, is comprised of a selection of a series of source files (`src()`), optionally performing some additional processing on each file (via `pipe()`) and then writing those files to a destination path (`dest()`). If no additional processing is specified (as in the prior example), Gulp will simply copy the files that match the wildcard pattern. The beauty of streams, however, is that one can execute any number of transformations before the final data is saved. So, the workflows can become very complex.

Now that you’ve seen a simple task, let’s get into some more complicated tasks in the next section.
Creating a modular Gulp configuration

Although you can add all the tasks that you want to run to a single configuration file, this quickly becomes unwieldy as you add more tasks to your environment. In order to keep your configuration maintenance easy, it’s best to split everything up into separate files.

This means that aside from our project's directory structure, our Gulp configuration has its own structure. The following shows how I like to structure my configuration:

```plaintext
project-root/
gulpfile.js            # Stub (loads in everything else)
gulp/                  #
cfg.js                 # Configuration - files to copy,
            # output paths, etc.
settings.js           # values of command-line flags
tasks.js              # Stub (loads in all the tasks)
tasks/                #
some-task.js          # Contains each task, in its own
another-task.js       # JavaScript file
utils/                #
paths.js              # Utility functions all tasks share
            # path manipulation methods
```

Let's go over the code that is in some of the above files. First, let's look at a simplified `gulp/config.js` file, which stores the base paths, as well as source and destination paths for our project:

```javascript
var config = {
  paths: {
    base: process.cwd(), // [1]
    dest: "build",       // [2]
    src: "src",          // [3]
    config: "config"     // [4]
  },
  assets: { // [5]
    copy: [ // [6]
      {src: "www/**.*", dest: "www"},
      {src: "www/html/**/*", dest: "www/html"},
      {src: "www/img/**/*", dest: "www/img"},
      {src: "www/js/lib/**/*", dest: "www/js/lib"},
      {src: "res/**/*", dest: "res"}
    ]
  }
},
module.exports = config; // [7]
```

This is a fairly simple configuration file—we'll end up adding much more to it as the book progresses.

The first section defines the various paths that Gulp will need to know in order to copy our project files as well as those necessary for transforming our code. The base path ([1]) is used as the foundation for every other path, and as such, every other path you see will be relative, not absolute.

The output directory is specified in [2], and the source directory is specified in [3]. Configuration files that we might need for code transformation and style checking are specified in [4]. Each one is relative to the base path.

Every project has a set of assets, and ours is no exception – these are specified in section [5]. In this case, we don't have very many, but even so, they need to be specified so that our tasks know what files they need to work with. We may have many different assets, some of which may require different processing, so we can add to this section as we need. For now, we just need to copy some files, and so we add them to the copy section ([6]). Notice that we specify them in terms of a source wildcard string and a destination path. These will automatically be made relative to the src ([3]) and dest ([2]) paths.

The final line ([7]) is used to export the information out of this file. We can then require the file later in another file (and most of our tasks and the like will do so). This means that our asset and path configuration only needs to be maintained in one place.

Gulp can accept custom command-line arguments, and these can be used to control how various tasks operate. A typical argument might specify the amount of logging that is generated. This is all handled by the gulp/settings.js file. Let's take a look:

```javascript
var gutil = require("gulp-util");
var settings = {
    VERBOSE: gutil.env.verbose ? (gutil.env.verbose === "yes") : false
};

module.exports = settings;
```

Right now, there's not a lot going on in this file, and that's because we really don't have tasks that need to be configured using command line arguments. But we'll be adding to this file as the book goes on.
By itself, this file doesn't do much. All it is doing is using `gutil.env` to read the arguments passed on the command line. In this case, it's checking to see if we passed `verbose` on the command line. If we did, and the value was `yes`, `settings.VERBOSE` would be set to `true`. If we didn't (or if we did and the value was `no`), `settings.VERBOSE` would be set to `false`. If we want to take advantage of this setting later on in a task, we can do so.

There's one other file in the `gulp/` directory, so let's take a look at `gulp/tasks.js`:

```javascript
var path = require("path");
var tasks = require("require-all") (path.join(__dirname, "tasks"));
module.exports = tasks;
```

As you can see, it's a very short file. All it does is find all the tasks within `gulp/tasks/` and load them into the `tasks` object. Right now that would return an empty object, but by the end of the chapter, the `tasks` object will contain several methods that Gulp can use. We use the `require-all` package to make life easier on us— that way we don't have to individually `require` each and every task. Later on, when we add additional tasks to our Gulp configuration, it means we don't have to later come back and edit this file.

Next, let's look at `gulp/utils/paths.js`:

```javascript
var path = require("path"),
    config = require("../config"); // [1]
function makeFullPath(filepath, relativeTo) {
    var pathComponents = [config.paths.base];
    if (relativeTo) {
        pathComponents.push(config.paths[relativeTo]);
    }
    pathComponents = pathComponents.concat(filepath.split("/"));
    return path.join.apply(path, pathComponents);
}
module.exports = {
    SRC: "src",
    DEST: "dest",
    CONFIG: "config",
    makeFullPath: makeFullPath
};
```

This utility file provides a mechanism our tasks can use to craft paths that are relative to the source, destination, configuration, and base paths in our project. It makes heavy use of Node.js' `path` library so that our Gulp tasks can work across different platforms.
Finally, we need to create the actual gulpfile.js file that kicks everything off. It doesn't do much on its own; instead it loads everything else in and configures any available tasks with Gulp:

```javascript
require("babel/register"); // [1]
var gulp = require("gulp"),
    tasks = require("./gulp/tasks"); // [2]
Object.keys(tasks).forEach(function(taskName) { // [3]
    var taskOpts = tasks[taskName];
    if (typeof taskOpts === "function") {
        gulp.task(taskName, taskOpts); // [4]
    } else {
        gulp.task(taskName, taskOpts.deps, taskOpts.task); // [5]
    }
});
```

The first line ([1]) imports Babel so that we can use ES2015 code in our tasks should we choose to. The third line ([2]) imports all the tasks that are available. Right now this will be an empty object, but as we add tasks, it will contain more and more functions that Gulp can use to copy and transform files.

The code starting at [3] just takes all the available tasks and creates a corresponding Gulp task. Each task can either be a function, or it can be an object that specifies dependencies (and more), hence the two different method invocations at [4] and [5].

### Copying assets

Now that we've created the basic Gulp configuration structure, let's create our first task to copy our app's assets from the source path to the destination path.

We can call this file gulp/tasks/copy-assets.js, and it should look like this:

```javascript
var merge = require("merge-stream"),
gulp = require("gulp"),
config = require("./config"),
paths = require("./utils/paths");

function copyAssets() {
    return merge.apply(merge, config.assets.copy.map(function(asset) {
        var fqSourcePath = paths.makeFullPath(asset.src, paths.SRC);
        var fqTargetPath = paths.makeFullPath(asset.dest, paths.DEST);
        return gulp.src([fqSourcePath])
            .pipe(gulp.dest(fqTargetPath));
    }));
}
```
Let's go over what this task is really doing:

- We're using our utility method paths.makeFullPath (which uses path.join) to ensure that our configuration works across multiple platforms. On Unix-like systems, the path separator is /; but on Windows systems, the path separator is actually \. In order to simplify the configuration, however, we're using / in config.assets.copy.makeFullPath.splits(\) each one of the strings into arrays, and uses path.join (which knows the correct path separator) to create the final path.

- map iterates over an array and returns a new array using a given transformation. For example, \[1, 2, 3\].map(function(x) {return x*2;}) will return a new array of \[2, 4, 6\]. In our case, we're returning an array of gulp.src(…).pipe(gulp.dest(…)) chains. We can then apply the array to merge in order to combine all the tasks together.

- apply is a way to call a function that accepts multiple arguments using an array instead. For example, console.log.apply(console,[1,2,3]) will log 1 2 3. This is different from console.log([1,2,3]) which instead will log [1, 2, 3].

At this point, you can type the following on the command line and copy the project assets from their source location to their destination:

```
$ gulp copy-assets
```

### Performing substitutions

Many times, we need to convert certain keywords in a Gulp stream into some other values. A simple example is to transform \{{version}\} your app's version number—for example, into \1.23.4456\. Doing this is pretty simple, but it opens up a large number of possibilities.
To do this, we’ll use the gulp-replace-task plugin. This plugin will replace all the instances of a particular regular expression with a replacement value. These expressions can become very complex; but in our case, we'll keep them simple.

We’ll only need to support substitutions in our code files, so let’s create a new task that is designed to copy our code files and apply any necessary substitutions along the way. We'll call it gulp/tasks/copy-code.js. The file should start as follows:

```javascript
var gulp = require("gulp"),
    replace = require("gulp-replace-task"),
    concat = require("gulp-concat"),
    pkg = require("../../package.json"),
    config = require("../config"),
    paths = require("../utils/paths");
```

Next, we need to define a method that will perform substitutions on the input streams. Remember, these will be the files matched by the pattern provided to gulp.src():

```javascript
function performSubstitutions() {
    return replace({
        patterns: [
            { 
                match: /{{{VERSION}}}/g,
                replacement: pkg.version
            }
        ]
    });
}
```

Next, let's define another configuration setting that specifies the code files that do need substitutions and where they should be stored. In gulp/config.js, add a code section to the config.assets object, like this:

```javascript
assets: {
    copy: [ ... ],
    code: {src: "www/js/app/**/*.js", dest: "www/js/app"}
}, ...
```

Next, we need to define the code that will copy the files specified by config.assets.code to the appropriate destination. This will be added to gulp/tasks/copy-code.js, and it should look like this:

```javascript
function copyCode() {
    return gulp.src([paths.makeFullPath(config.assets.code.src, paths.SRC)])
```
Task Automation

```javascript
.pipe(performSubstitutions())
.pipe(concat("app.js"))
.pipe(gulp.dest(paths.makeFullPath(
    config.assets.code.dest, paths.DEST)));
```

```javascript
module.exports = {
    task: copyCode
}
```

The `copyCode` method is pretty simple to follow. First, all the JavaScript files are located using the configuration we've specified. These are all passed through `performSubstitutions()`. The results of the substitutions are then packaged together in a neat little bundle with `concat`. So, even if we have multiple JavaScript files, they will all be packaged into a single file (`app.js`).

You don't have to concatenate your files if you don't want to. When you have multiple JavaScript files, however, it means that you have to include each one in your `index.html` file. Whereas if you bundle them into a single file, you reduce the number of `script` tags you have in your `index.html` file.

To test these tasks, we can create two simple files. The first should be placed in `src/www/` and named `index.html`:

```html
<!DOCTYPE html>
<html>
    <head>
        <script src="cordova.js" type="text/javascript"></script>
        <script src="js/app/app.js" type="text/javascript"></script>
    </head>
    <body>
        <p>Hello!</p>
        <div id="demo"></div>
    </body>
</html>
```

The second file should be in `src/www/js/app/` and named `index.js`:

```javascript
document.getElementById("demo").textContent = "{{{VERSION}}}";
```
The JavaScript file itself is very simple, obviously. The idea is simply to prove that our Gulp tasks work. If you execute `gulp copy-assets`, you'll find that `index.html` has been copied from `src/www/` to `build/www/`. Likewise, if you execute `gulp copy-code`, you'll find that `index.js` has been copied from `src/www/js/app/` to `build/www/js/app/` and renamed to `app.js`. If you open the latter file in an editor, you'll also see that `{{{VERSION}}}` has been replaced with `1.0.0` (which came from package.json).

As you may recall, we indicated earlier in this chapter that we still need a `config.xml` file. This is true, but we've specified everything we need in `package.json`. Wouldn't it be great to generate a valid `config.xml` file from a template? This means that we need more substitutions and a proper template.

Let's define our template first. This should be in `src/config.xml` (see the code package for the entire file):

```xml
<?xml version='1.0' encoding='utf-8'?
<widget id='{{{ID}}}' version='{{{VERSION}}}'
xmlns='http://www.w3.org/ns/widgets'
xmlns:cdv='http://cordova.apache.org/ns/1.0'
xmlns:gap='http://phonegap.com/ns/1.0'>
 <name> {{{NAME}}} </name>
 <description>
  {{{DESCRIPTION}}} 
 </description>
 <author email='{{{AUTHOR.EMAIL}}}'href='{{{AUTHOR.SITE}}}'>
  {{{AUTHOR.NAME}}} 
 </author>
 <content src='index.html' />
  {{{PREFS}}} 
 <access origin='*' />

 ... 
</widget>
```

Notice that there are a lot of substitution variables in the preceding code. Most of them are pretty simple: `{{{ID}}}`, `{{{NAME}}}`, and so on. One of them is a little more complex: `{{{PREFS}}}`. This will need to render our simpler list of preferences in `package.json` into the XML format required by Cordova.

Let's create a new utility file named `gulp/utils/performSubstitutions.js` with a new version of the `performSubstitutions` method. We'll need this new version in two tasks, hence the need to split it out into its own file. The new file should look like this:

```javascript
var pkg = require("../package.json"),
    replace = require("gulp-replace-task");
```
function performSubstitutions() {
    function transformCordovaPrefs() {
        var template = '<preference name="{{{NAME}}}"' + 
            'value="{{{VALUE}}}" />';
        if (pkg.cordova &&
            pkg.cordova.preferences instanceof Object) {
            return Object.keys(pkg.cordova.preferences).map(
                function(prefName) {
                    var str = template.replace(/{{{NAME}}}/g,
                        prefName)
                        .replace(/{{{VALUE}}}/g,
                            pkg.cordova.preferences[prefName]);
                    return str;
                }).join("\n  ");
        }
    }

    return replace({
        patterns: [
            {
                match: /{{{VERSION}}}\g,  
                replacement: pkg.version
            },
            {
                match: /{{{ID}}}\g,    
                replacement: pkg.cordova.id
            },
            {
                match: /{{{NAME}}}\g,    
                replacement: pkg.cordova.name
            },
            {
                match: /{{{DESCRIPTION}}}\g,  
                replacement: pkg.cordova.description
            },
            {
                match: /{{{AUTHOR.NAME}}}\g,  
                replacement: pkg.cordova.author.name
            },
            {
                match: /{{{AUTHOR.EMAIL}}}\g,  
                replacement: pkg.cordova.author.email
            }
        ]
    });
}
Next, we'll need to edit gulp/copy-code.js to include this new version. Remove the `performSubstitutions` method from this file first, and then add the following require to the top of the file:

```javascript
var ...,
    performSubstitutions = require("./utils/performSubstitutions");
```

Finally, let's add another task that can copy the configuration file. We'll call it `gulp/tasks/copy-config.js`, and it should look like this:

```javascript
var gulp = require("gulp"),
    performSubstitutions =
        require("./utils/performSubstitutions"),
    config = require("./config"),
    paths = require("./utils/paths");

function copyConfig() {
    return gulp.src([paths.makeFullPath("config.xml", paths.SRC)])
        .pipe(performSubstitutions())
        .pipe(gulp.dest(paths.makeFullPath(".",
                                        paths.DEST))));
}

module.exports = {
    task: copyConfig
}
```

Of course, we don't want to have to run lots of individual tasks just to copy files. So let's create a simple task that depends upon these three tasks. By doing so, Gulp will run all of these tasks with a single command.
Let's create the new task with the name gulp/tasks/copy.js. The file should contain the following:

```javascript
module.exports = {
    deps: ["copy-assets", "copy-config", "copy-code"],
}
```

This is the shortest task so far. All it does is list the other three tasks as dependencies. This means that they will be executed prior to copy. Since copy doesn't contain any additional code, it's just a simple way to execute several tasks at once. If you execute gulp copy, you'll find that you have a new config.xml file under build. It should look a lot like the following:

```xml
<?xml version='1.0' encoding='utf-8'?>
<widget id="com.packtpub.logologyv1" version="1.0.0"
    xmlns="http://www.w3.org/ns/widgets"
    xmlns:cdv="http://cordova.apache.org/ns/1.0"
    xmlns:gap="http://phonegap.com/ns/1.0">
    <name>Logology</name>
    <description>
        Dictionary application for Mastering PhoneGap book
    </description>
    <author email="kerrishotts@gmail.com"
        href="http://www.photokandy.com">
        Kerri Shotts
    </author>
    <content src="index.html" />
    <preference name="permissions" value="none" />
    <preference name="fullscreen" value="false" />
    <preference name="orientation" value="default" />
    ...
    <access origin="*" />
</widget>
```

Now that you've mastered the method of performing substitutions, you will learn how to interact with Cordova programmatically in the next section.

**How to execute Cordova tasks**

It's tempting to use the Cordova command-line interface directly, but there's a problem with this: there's no great way to ensure what you write works across multiple platforms. If you are certain you'll only work with a specific platform, you can go ahead and execute shell commands instead; but what we're going to do is a bit more flexible.
The code in this section is inspired by https://github.com/kamrik/CordovaGulpTemplate.

The Cordova CLI is really just a thin wrapper around the cordova-lib project. Everything the Cordova CLI can do, cordova-lib does as well.

Because the Cordova project will be a build artifact, we need to be able to create a Cordova project in addition to building the project. We'll also need to emulate and run the app. To accomplish this, we'll need to create a new utility file named gulp/utils/cordova-tasks.js. At the top we require cordova-lib and other packages we'll need:

```javascript
var cordovaLib = require("cordova-lib"),
    pkg = require("../package.json"),
    config = require("../config"),
    path = require("path"),
    settings = require("../settings"),
    paths = require("../utils/paths"),
    cordova = cordovaLib.cordova.raw;

Next, let's create the code to create a new Cordova project in the build directory:

```javascript
var cordovaTasks = {
    create: function create() {
        return cordova.create(paths.makeFullPath(".", paths.DEST),
            pkg.cordova.id, pkg.cordova.name,
            {
                lib: {
                    www: {
                        url: path.join(process.cwd(), pkg.cordova.template),
                        link: false
                    }
                }
            }
        );
    }
};
```

module.exports = cordovaTasks;
```
Although it’s a bit more complicated than cordova create is on the command line, you should be able to see the parallels. The lib object is passed simply to provide a template for the project (equivalent to --copy-from on the command line). In our case, package.json specifies that this should come from the blank/ directory in the code bundle of this book. If we don’t do this, all our apps would be created with the sample Hello World app that Cordova installs by default.

Our blank project template resides in ../blank relative to the project root. Yours may reside elsewhere (since you’re apt to reuse the same template), so package.json can use whatever path you need. Or, you might want the template to be within your project’s root; in which case, package.json should use a path inside your project’s root directory.

We won’t create a task to use this just yet. We need to define several other methods to build and emulate the Cordova app. First, we need to add some additional settings to gulp/settings.js:

```javascript
var settings = { ...
    PLATFORM = gutil.env.platform ? gutil.env.platform : "ios",
    BUILD_MODE = gutil.env.mode ? gutil.env.mode : "debug",
    BUILD_PLATFORMS = (gutil.env.for ? gutil.env.for
                      : "ios,android").split(",")
                   .filter(x => x),
    TARGET_DEVICE = gutil.env.target ? "--target=" +
                      gutil.env.target : ""
}
```

Next, let’s continue to add the additional methods we need to the cordovaTasks object:

```javascript
var cordovaTasks = {
    create: function create() {
        /* as above */
    },
    cdProject: function cdProject() {
        process.chdir(paths.makeFullPath("www", paths.DEST));
    },
    cdUp: function cdUp() {
        process.chdir("..");
    },
    // cordova plugin add ...
    addPlugins: function addPlugins() {
        cordovaTasks.cdProject();
        return cordova.plugins("add", pkg.cordova.plugins)
            .then(cordovaTasks.cdUp);
    },
};
```
addPlatforms: function addPlatforms() {
  cordovaTasks.cdProject();
  function transformPlatform(platform) {
    return path.join(process.cwd(), "node_modules", "cordova-" + platform);
  }
  return cordova.platforms("add", pkg.cordova.platforms.map(transformPlatform))
    .then(cordovaTasks.cdUp);
},

build: function build() {
  var target = settings.TARGET_DEVICE;
  cordovaTasks.cdProject();
  if (!target || target === "" || target === "--target=device") {
    target = "--device";
  }
  return cordova.build({
    platforms: settings.BUILD_PLATFORMS,
    options: ["--" + settings.BUILD_MODE, target]
  }).then(cordovaTasks.cdUp);
},

emulate: function emulate() {
  cordovaTasks.cdProject();
  return cordova.emulate({
    platforms: [settings.PLATFORM],
    options: ["--" + settings.BUILD_MODE, settings.TARGET_DEVICE]
  }).then(cordovaTasks.cdUp);
},

run: function run() {
  cordovaTasks.cdProject();
  return cordova.run({
    platforms: [settings.PLATFORM],
    options: ["--" + settings.BUILD_MODE, "--device", settings.TARGET_DEVICE]
  }).then(cordovaTasks.cdUp);
},

init: function() {
return cordovaTasks.create()
    .then(cordovaTasks.copyConfig)
    .then(cordovaTasks.addPlugins)
    .then(cordovaTasks.addPlatforms);
}]


Most of the previous tasks should be fairly self-explanatory; they correspond directly to their Cordova CLI counterparts. A few, however, need a little more explanation:

- **cdProject / cdUp**: These change the current working directory. All the cordova-lib commands after create need to be executed from within the Cordova project directory, not our project’s root directory. You should notice them in several of the tasks.
- **addPlatforms**: The platforms are added directly from our project's dependencies, rather than from the Cordova CLI. This allows us to control the platform versions we are using. As such, addPlatforms needs to do a little more work to specify the actual directory name of each platform.
- **build**: This executes the cordova build command. By default, CLI builds every platform. Since we will want to control the platforms that are built, hence we can use BUILD_PLATFORMS to control this behavior. On iOS, the build for an emulator is different than the build for a physical device. So, we also need a way to specify this, which is what TARGET_DEVICE does. This will look for emulators with the name specified for TARGET_DEVICE. But we might want to build for a physical device; in which case, we will look for device (or no target specified at all) and switch over to the --device flag which forces Cordova to build for a physical device.
- **init**: This does the hard work of creating the Cordova project, copying the configuration file (and performing substitutions), adding plugins to the Cordova project, and then adding platforms.
Now is also a good time to mention that we can specify various settings with switches on the Gulp command line. In the earlier snippet, we’re supporting the use of `--platform` to specify the platform to emulate or run, `--mode` to specify the build mode (debug or release), `--for` to determine what platforms Cordova will build for, and `--target` to specify the target device. The code will specify reasonable defaults if these switches aren't specified; but they also allow the developer extra control over the workflow, which is very useful. For example, we'll be able to use commands like the following:

```
$ gulp build --for ios,android --target device
$ gulp emulate --platform ios --target iPhone-6s
$ gulp run --platform ios --mode release
```

Next, let's write the code to actually perform various Cordova tasks. It isn't difficult, but we need to create a lot of files. Each file name in the code below is in comments:

```
// gulp/tasks/clean.js
var paths = require("./utils/paths"),
    config = require("./config"),
    rimraf = require("rimraf");
function clean(cb) {
    var BUILD_PATH = paths.makeFullPath(".", paths.DEST);
    rimraf(BUILD_PATH, cb);
}
module.exports = {
    task: clean
}

// gulp/tasks/init.js
var cordovaTasks = require("./utils/cordova-tasks");
module.exports = {
    deps: ["clean"],
    task: cordovaTasks.init
};

// gulp/tasks/build.js
var cordovaTasks = require("./utils/cordova-tasks");
module.exports = {
    deps: ["copy"],
    task: cordovaTasks.build
};

// gulp/tasks/emulate.js
var cordovaTasks = require("./utils/cordova-tasks");
```
There's a catch with the `init` task: it will fail if anything is already in the `build/` directory. As you can guess, this could easily happen; so we also created a `clean` task. This uses `rimraf` to delete a specified directory, which is equivalent to using `rm -rf build`. We then ensured that `init` depends on `clean`. So, whenever we execute `gulp init`, the old Cordova project is removed and a new one is created for us.

Finally, note that all the `build` (and other) tasks depend on `copy`. This means that all our files in `src/` will be copied (and transformed, if necessary) to `build/` prior to executing the desired Cordova command. As you can see, our tasks are already becoming very complex, while remaining comprehensible when they are taken singularly.

This means that we can now use the following tasks in Gulp:

```
$ gulp init                   # create the cordova project;
                           # cleaning first if needed
$ gulp clean                # remove the cordova project
$ gulp build                # copy src to build; apply
                           # transformations; cordova build
$ gulp build --mode release # do the above, but build in
                           # release mode
$ gulp build --for ios      # only build for iOS
$ gulp build --target=device # build device versions instead of
                           # emulator versions
$ gulp emulate --platform ios # copy src to build; apply
                           # transformations;
                           # cordova emulate ios
$ gulp emulate --platform ios --target iPhone-6 # same as above, but open the
                           # iPhone 6 emulator
```
$ gulp run --platform ios     # copy src to build;
# apply transformations;
# cordova run ios --device

Now, you're welcome to use the previous code as it is or you can use an NPM package that takes care of the `cordovaTasks` portion for you. This has the benefit of drastically simplifying your Gulp configuration. We've already included this package in our `package.json` file as well as our Gulp configuration. It's named `cordova-tasks` and was created by the author. It shares a lot of similarities to the earlier code. To see how it works (and how much simpler the tasks become), see `logology-v01/gulp` in the code package for this book.

This was one of the complex sections; so if you've come this far, take a coffee break. Next, we'll worry about managing app version numbers.

## Managing version numbers

Although we've set up our `copy-config` and `copy-code` tasks to substitute the version number whenever `{{VERSION}}` is encountered, we don't have any tasks that actually change the version. We could just edit `package.json`, of course. But this is tedious and it can't be included automatically in any other Gulp task. Instead, let's use the `gulp-bump` plugin to take care of this for us.

`gulp-bump` is a very simple plugin: it is designed to take a `package.json` (or similar) file and edit the `version` property based on specific commands. Most versions are of the `major.minor.patch` form and we can ask it to increment any portion by one. If you wanted, you could increment the `patch` portion of the version to automatically track build numbers, for example.

Doing this is pretty simple. Let's first create another utility file, this time called `gulp/utils/bump.js`:

```javascript
var gulp = require("gulp");
var bump = require("gulp-bump");

module.exports = function bump(importance) {
  return gulp.src([path.join(process.cwd(), "package.json")])
    .pipe(bump({type: importance}))
    .pipe(gulp.dest(process.cwd()));
}
```
Task Automation

The importance variable can be one of the following strings: major, minor, or patch. Next, let's create three tasks that will allow us to call this method directly (again, these are in three separate files, indicated in the comment):

```javascript
// gulp/tasks/version-bump-patch.js
var bump = require("../utils/bump");
module.exports = {
    task: bump.bind(null, "patch")
}

// gulp/tasks/version-bump-minor.js
var bump = require("../utils/bump");
module.exports = {
    task: bump.bind(null, "minor")
}

// gulp/tasks/version-bump-major.js
var bump = require("../utils/bump");
module.exports = {
    task: bump.bind(null, "major")
}
```

Now you can directly bump the version number by executing `gulp version-bump-patch`. This, however, only edits `package.json`. If you want the files in `build/` to reflect this, you will need to also execute `gulp copy` (or `build` and so on).

Supporting ES2015

We've already mentioned ES2015 (or EcmaScript 2015) in this chapter. Now is the moment to start using it. First, though, we need to modify our `copy-code` task to transpile from ES2015 to ES5, or our code will not run on any browser that doesn't support the new syntax (which is still quite a few mobile platforms).

There are several transpilers available. I prefer Babel (https://babeljs.io).

We used Babel 5.x. Although Babel 6 has recently been released, as of this writing, the demonstration app and corresponding Gulp configurations have not been updated to Babel 6.x.
There is a Gulp plugin that we can use, which makes this transpilation transform extremely simple. To do this, we need to add the following to the top of gulp/tasks/copy-code.js:

```javascript
var ...,  
babel = require("gulp-babel"),  
sourcemaps = require("gulp-sourcemaps"),  
gutil = require("gulp-utils");
```

Source maps are an important piece of the debugging puzzle. As our code will be transformed by the time it runs on our device, debugging could become a little more difficult, since the line numbers and the like don't match. Sourcemaps provide the browser with a map between your ES2015 code and the final result so that debugging is a lot easier.

Next, let's modify our `projectTasks.copyCode` method:

```javascript
function copyCode() {
    var isRelease = (settings.BUILD_MODE === "release");
    return gulp.src([paths.makeFullPath(config.assets.code.src, 
                     paths.SRC)])
        .pipe(cordovaTasks.performSubstitutions())
        .pipe(settings.BUILD_MODE === "debug" ? 
             sourcemaps.init() : gutil.noop())
        .pipe(babel())
        .pipe(concat("app.js"))
        .pipe(settings.BUILD_MODE === "debug" ? 
             sourcemaps.write() : gutil.noop())
        .pipe(gulp.dest(paths.makeFullPath(
                         config.assets.code.dest, paths.DEST)));
}
```

Our task is now a little more complex; but this is only because we want to control when source maps are generated. When `babel()` is called, it will convert the ES2015 code to ES5 and also generate a sourcemap of the changes. This makes debugging easier, but it also increases the file size (sometimes by quite a bit). As such, when we're building in release mode, we don't want to include the sourcemaps. So, we will call `gutil.noop` instead, which will just do nothing.

The sourcemap functionality requires us to call `sourcemaps.init` prior to any Gulp plugin that might generate sourcemaps. After the plugin that creates the sourcemaps is executed, we also have to call `sourcemaps.write` to save the sourcemap back in the stream. We could also write the sourcemap to a separate `.map` file by calling `sourcemaps.write(".")`. But you would need to be careful about cleaning up that file while creating a release build.
babel does the actual hard work of converting ES2015 code into ES5. But it does need a little help in the form of a small support library. We'll add this library to src/www/js/lib/ by copying it from the gulp-babel module:

```
$ cp node_modules/babel-core/browser-polyfill.js src/www/js/lib
```

If you don't have the src/www/js/lib/directory yet, you'll need to create it before you execute the previous command.

Next, we need to edit src/www/index.html to include this script. While we're at it, let's make a few other changes:

```html
<!DOCTYPE html>
<html>
<head>
  <script src="cordova.js" type="text/javascript"></script>
  <script src="./js/lib/browser-polyfill.js" type="text/javascript"></script>
  <script src="./js/app/app.js" type="text/javascript"></script>
</head>
<body>
  <p>This is static content..., but below is dynamic content.</p>
  <div id="demo"></div>
</body>
</html>
```

Finally, let's write some ES2015 code in src/www/js/app/index.js:

```javascript
function h(elType, ...children) {
  let el = document.createElement(elType);
  for (let child of children) {
    if (typeof child !== "object") {
      el.textContent = child;
    } else if (child instanceof Array) {
      child.forEach(el.appendChild.bind(el));
    } else {
      el.appendChild(child);
    }
  }
  return el;
}

function startApp() {
```
This chapter isn't about how to write the ES2015 code, so I won't bore you with all the details. We'll cover that in the next chapter. Suffice it to say, the earlier code generates a few list items when the app is run using a very simple form of DOM templating. But it does so using the ... (spread) syntax for variable parameters: the for ... of loop and let instead of var. Although it looks a lot like JavaScript, ES2015 is different enough that it will take some time to learn how best to use the new features.

**Linting your code**

You could execute `gulp emulate --platform ios` (or `android`) right now; the app should work. But how do we know whether our code will work when it is built? Better yet—how can we prevent a build if the code isn't valid?

We do this by adding lint tasks to our Gulp configuration. Linting is a lot like compiling; the linter checks your code for obvious errors and aborts if it finds any. There are various linters available (some better than others), but not all of them support ES2015 syntax yet. The best one that does is ESLint (http://www.eslint.org). Thankfully, there's a very simple Gulp plugin that uses it.

We could stop at linting and be done, but code style is also important, and can catch out potentially serious issues as well. As such, we're also going to be using the JavaScript Code Style checker or JSCS (https://github.com/jscs-dev/node-jscs).

Let's create tasks to lint and check our coding style. First, we need to add some additional configuration to gulp/config.js:

```javascript
var config = { ...
    lint: "src/www/js/app/**/*.js",
    "code-style": "src/www/js/app/**/*.js"
};
```
Now, let’s create the associated tasks. Let’s start with linting – this will live in the gulp/tasks/link.js file:

```javascript
var gulp = require("gulp"),
eslint = require("gulp-eslint"),
config = require("./config"),
settings = require("./settings"),
paths = require("./utils/paths");
function lintCode() {
    return gulp.src(paths.makeFullPath(config.lint))
        .pipe(eslint(paths.makeFullPath("eslint.json",
            paths.CONFIG)))
        .pipe(eslint.format());
}
module.exports = {
    task: lintCode
}
```

The task for checking our code style will be named gulp/tasks/code-style.js. It should have the following code:

```javascript
var gulp = require("gulp"),
jscs = require("gulp-jscs"),
config = require("./config"),
settings = require("./settings"),
paths = require("./utils/paths");
function checkCodeStyle() {
    return gulp.src(paths.makeFullPath(config['code-style']))
        .pipe(jscs({
            configPath: paths.makeFullPath("jscs.json",
                paths.CONFIG),
            esnext: true
        }))
        .pipe(jscs.reporter())
        .pipe(jscs.reporter('fail'));
}
module.exports = {
    task: checkCodeStyle
}
```

Now, before you run either task, you'll need two configuration files to tell each task what should be an error and what shouldn't be. We suggest using the files from the code bundle for this chapter for now (you can find them in the logology-v01/config directory). If you want to change the settings, you can do so; the sites for ESLint and JSCS both have information on how to modify the configuration files.
Chapter 1

config/eslint.json must contain "parser": "babel-eslint" in order to force it to use the ES2015 syntax. This is set for JSCS in the Gulp configuration, however.

config/jscs.json must exist and must not be empty. If you don't need to specify any rules, use an empty JSON object ({}).

Now, if you were to execute gulp lint and our source code had a syntax error, you would receive an error message. The same goes for code styles; gulp code-style would generate an error if it didn't like the look of the code.

Next, you should add these two tasks to our build, emulate, and run tasks. Here's what the module.exports of gulp/tasks/build.js looks like after doing this:

```javascript
module.exports = {
  deps: ["copy", "lint", "code-style"],
  task: ...
}
```

Now, if you execute gulp build and there is a linting or code style error, the build will fail with an error. This gives a little more assurance that our code is at least syntactically valid prior to distributing or running the code.

Linting and style checks do not guarantee that your code works logically. It just ensures that there are no syntax or style errors. If your program responds incorrectly to a gesture or processes some data incorrectly, a linter won't necessarily catch these issues.

Uglifying your code

Code uglification or minification sounds a bit painful, but it's a really simple step we can add to our workflow. It will reduce the size of our applications when we build in release mode. Uglification also tends to obfuscate our code a little bit, but don't rely on this for any security—obfuscation can be easily undone.

To add code uglification, add the following line of code to the top of our gulp/tasks/copy-code.js file:

```javascript
var ...,
  uglify = require("gulp-uglify");
```
We can then uglify our code by adding the following code immediately after
`.pipe(concat("app.js"))` in our `projectTasks.copyCode` method:

```
.pipe(isRelease ? uglify({preserveComments: "some"}) :
gutil.noop())
```

Notice that we added the `uglify` method only when the build mode was `release`. This means that we'll only trigger it if we execute `gulp build --mode release`.

You can, of course, specify additional options. If you want to see all the
documentation, visit https://github.com/mishoo/UglifyJS2/. Our options
include certain comments (which most likely are license-related) while stripping out all the other comments.

### Putting it all together

You've accomplished quite a bit, but there's one last thing we want to mention:
the default task. If `gulp` is run with no parameters, it will look for a default task
to perform. This can be anything you want.

To specify this, just add the following task at `gulp/tasks/default.js`:

```
module.exports = {
  deps: ["build"]
}
```

Now, if you execute `gulp` with no specific task, you'll actually start the `build` task
instead. What you want to use for your default task is largely dependent upon
your preferences.

Your Gulp configuration is now quite large and complex. We've added a few
additional features to it in the code package, so it wouldn't be a bad idea to take a
look at our final version in the code bundle of this book. We've also added several
other features to the configuration, which you might want to investigate further.
They are as follows:

- BrowserSync for rapid iteration and testing
- The ability to control whether or not any errors prevent further tasks from
  being executed
- Help text
Summary
In this chapter, you've learned why a task runner is useful and how to install Gulp and create several tasks of varying complexity to automate the building process of your project and other useful tasks. We're not yet done with Gulp, though. In the next chapter, we'll talk about modularization using Browserify and how to integrate it within our build tasks.
Where to buy this book

You can buy Mastering PhoneGap Mobile Application Development from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.