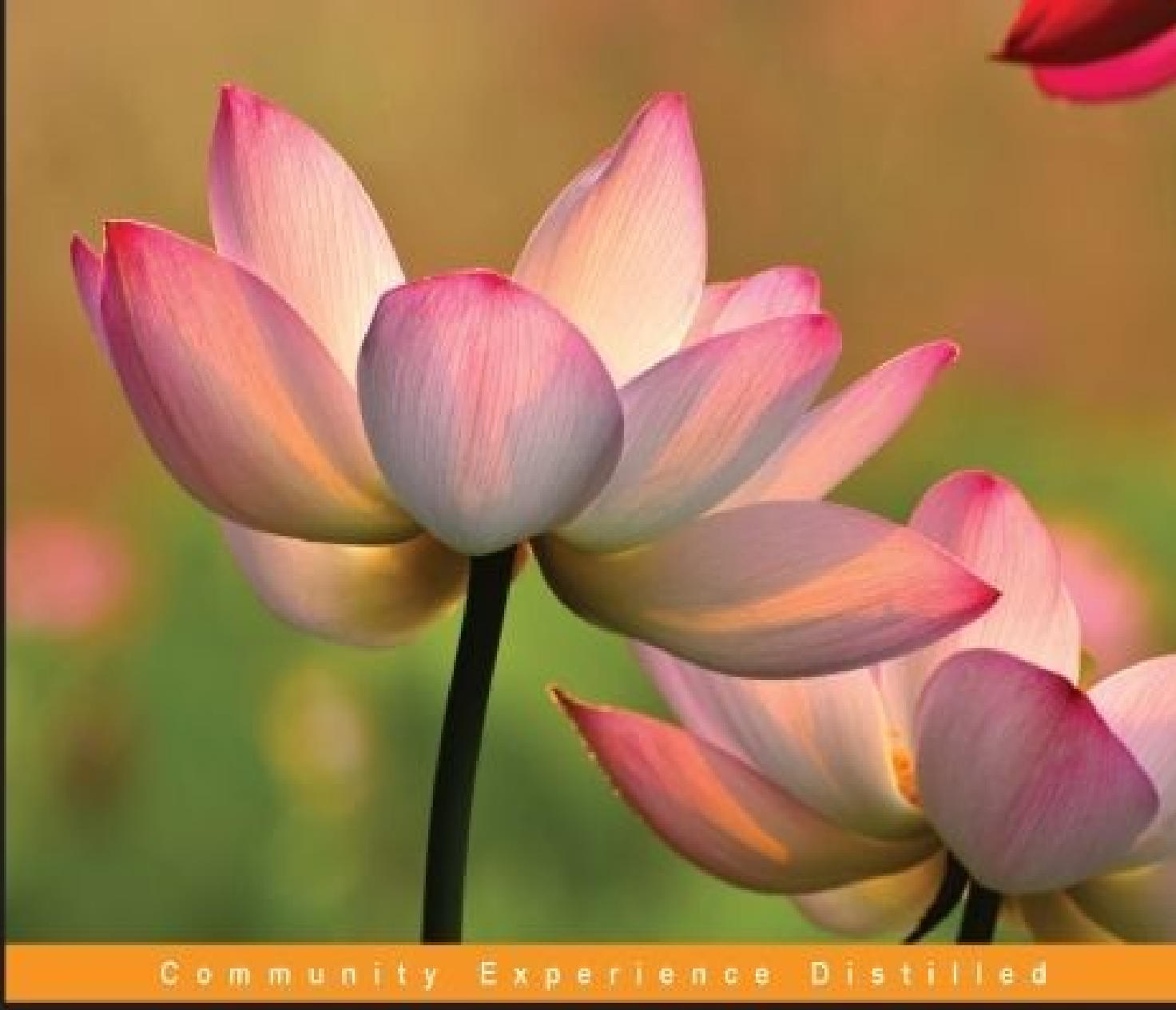


Community Experience Distilled

WebStorm Essentials

Build efficient HTML, CSS, and JavaScript applications using the powerful WebStorm IDE

Stefan Rosca Den Patin

[PACKT] open source
PUBLISHING

WebStorm Essentials

Table of Contents

WebStorm Essentials

WebStorm Essentials

First published: October 2015

Production reference: 1131015

Published by Packt Publishing Ltd.

Livery Place

35 Livery Street

Birmingham B3 2PB, UK.

ISBN 978-1-78528-695-7

www.packtpub.com

Credits

Authors

Stefan Rosca

Den Patin

Reviewer

Sonal Raj

Commissioning Editor

Priya Singh

Acquisition Editor

Reshma Raman

Content Development Editor

Amey Varangaonkar

Technical Editor

Vijin Boricha

Copy Editors

Shruti Iyer

Sonia Mathur

Project Coordinator

Nidhi Joshi

Proofreader

Safis Editing

Indexer

Monica Ajmera Mehta

Graphics

Disha Haria

Production Coordinator

Nilesh Mohite

Cover Work

Nilesh Mohite

About the Authors

Stefan Rosca is a Senior Front End Engineer with a passion for JavaScript on the client and the server. Likes to tweak things, loves a good challenge and benchmarks everything.

During the last years he specializes in front end development and worked on several projects at Word of Social, SynerTrade and VisualDNA.

He can be reached at http://uk.linkedin.com/in/roscastefan/.

I would like to thank my beautiful wife, Liliana, for her support and patience.

My parents deserve special thanks for exposing me to the world of computers from an early age.

I would also like to give special thanks to Velvet.

Den Patin is a senior software engineer with a hankering for writing clean code and creating elaborate yet user-friendly web applications. For over four years at and outside work, he has been utilizing a vast array of technologies for both frontend (HTML, CSS, JavaScript, CoffeeScript, jQuery, and AngularJS) and backend (Node.js, Ruby, and PHP).

Den holds a master's degree of science in applied informatics and a bachelor's degree of science in computer science and engineering and specializes in computational linguistics. His hobbies, besides programming, comprise of studying and teaching foreign languages and playing the piano.

During the last couple of years, Den developed and tailored various navigation and location-based web applications at T-Systems RUS.

To get in touch with him, you can visit his website, http://dpat.in.

Firstly, I would like to thank my parents, Marina and Valera, for their patience and constant encouragement during the authoring period, which was stressful yet fulfilling.

A humongus thanks goes to the Packt editorial team, especially to Amey Varangaonkar, for the assistance provided at any time and the highly professional managing and approach to the entire authoring process and Reshma Raman, for simply considering me to be an author one fine day and successfully getting the book off the ground.

The guys from JetBrains, the father of WebStorm and other IDEs, deserve special thanks for their powerful masterpieces that are too great for words.

Last but not least, I would like to thank my workfellows, Dima Streltsov and Artem Petrov, for their friendship and for facilitating the completion of my high workload.

About the Reviewer

Sonal Raj is a hacker, pythonista, big data believer, and technology dreamer. He has a passion for design and is an artist at heart. Sonal blogs about technology, design, and gadgets at http://www.sonalraj.com/. When not working on projects, he can be found travelling, stargazing, or reading.

He has pursued engineering in computer science and loves to work on community projects. He works extensively on graph computations using Neo4j and Storm and is a keen full stack web developer. Sonal has been a speaker at PyCon India and other meets and has also published articles and research papers in leading magazines and international journals. He has made several open source contributions.

He is the author of *Neo4j High Performance*, *Packt Publishing*, and has reviewed titles on Storm and Neo4j

I am grateful to the authors for patiently listening to my critique and I'd like to thank the open source community for keeping their passions alive and contributing to such remarkable projects. Special thanks to my parents without whom I never would have grown to love learning as much as I do.

www.PacktPub.com

Support files, eBooks, discount offers, and more

For support files and downloads related to your book, please visit www.PacktPub.com.

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.

https://www2.packtpub.com/books/subscription/packtlib

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can search, access, and read 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 a 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 9 entirely free books. Simply use your login credentials for immediate access.

Preface

Exploit the functional power of WebStorm to build better JavaScript applications!

Developers call WebStorm “the smartest JavaScript IDE”, and this couldn’t be any truer. It outperforms all its competitors and offers some very interesting features such as intelligent code assistance, debugging and testing, seamless integration of various tools, VCS integration, and many more.

This book is aimed at helping the reader get the benefit of all the features and possibilities that WebStorm brings to the world of frontend development.

We will go through all current technologies, building everything from simple websites to single page applications with Angular and Meteor and native mobile applications using HTML5, JavaScript, and CSS. We will also go through some of WebStorm’s power features that will help us boost our productivity and enhance developer experience!

What this book covers

Chapter 1, *Getting Started with WebStorm*, tells you concisely about WebStorm 10 and its new features. It helps you install it, guides you through its workspace, discusses setting up a new project, familiarizes you with the interface and useful features, and describes the ways to customize them to suit your needs.

Chapter 2, *Improving Your Efficiency with Smart Features*, exposes the most distinctive features of WebStorm, which are at the core of improving your efficiency in building web applications.

Chapter 3, *Developing Simple Web Pages*, describes the process of setting up a new project with the help of templates by importing an existing project, serving a web application, and using File Watchers.

Chapter 4, *Using Package Managers and Build Automation Tools*, describes using package managers and building systems for your application by means of WebStorm's built-in features.

Chapter 5, *AngularJS, React, Express, and Meteor – Developing Your Web Application*, focuses on the state-of-the-art technologies of the web industry and describes the process of building a typical application in them using the power of WebStorm features.

Chapter 6, *Immersing Yourself in Mobile App Development*, shows you how to use JavaScript, HTML, and CSS to develop a mobile application and how to set up the environment to test run this mobile application.

Chapter 7, *Analyzing and Debugging Your Code*, shows how to perform the debugging, tracing, profiling, and code style checking activities directly in WebStorm.

Chapter 8, *Testing Your Applications*, presents a couple of proven ways to easily perform application testing in WebStorm using some of the most popular testing libraries.

Chapter 9, *Getting to Know Additional yet Powerful Features*, is about a second portion of powerful features provided within WebStorm. In this chapter, we focus on some of WebStorm's power features that help us boost productivity and developer experience.

What you need for this book

This book will guide you through the installation of all the tools that you need to follow the examples. You will need to install WebStorm version 10 to effectively run the code samples present in this book.

Who this book is for

This book is intended for web developers with no knowledge of WebStorm yet but who are experienced in JavaScript, Node.js, HTML, and CSS and reasonably familiar with frameworks such as AngularJS and Meteor.

Conventions

In this book, you will find a number of text styles 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: “Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system.”

A block of code is set as follows:

```
html, body, #map {
    height: 100%;
    margin: 0;
    padding: 0
}
```

Any command-line input or output is written as follows:

```
$ mkdir css
$ cd css
```

New terms and **important words** are shown in bold. Words that you see on the screen, for example, in menus or dialog boxes, appear in the text like this: “The shortcuts in this book are based on the **Mac OS X 10.5+** scheme.”

Note

Warnings or important notes appear in a box like this.

Tip

Tips and tricks appear like this.

For this book we have outlined the shortcuts for the Mac OX platform if you are using the Windows version you can find the relevant shortcuts on the WebStorm help page https://www.jetbrains.com/webstorm/help/keyboard-shortcuts-by-category.html.

Reader feedback

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

To send us general feedback, simply e-mail <feedback@packtpub.com>, and mention the book's title in the subject of your message.

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 at 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 from your account at http://www.packtpub.com for all the Packt Publishing books you have purchased. 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 could 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/submit-errata, 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 to our website or added to any list of existing errata under the Errata section of that title.

To view the previously submitted errata, go to https://www.packtpub.com/books/content/support and enter the name of the book in the search field. The required information will appear under the **Errata** section.

Piracy

Piracy of copyrighted 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

If you have a problem with any aspect of this book, you can contact us at <questions@packtpub.com>, and we will do our best to address the problem.

Chapter 1. Getting Started with WebStorm

WebStorm is a full-fledged JavaScript **integrated development environment** (**IDE**) engineered by JetBrains for client-side and server-side web development. It is built on the basis of IntelliJ IDEA; WebStorm inherits all its features and peculiarities with which you can work conveniently with web-oriented languages, such as JavaScript, HTML, and CSS, as well as utilize essential-for-developers capabilities, such as refactoring, debugging, and version control management. Besides, WebStorm natively supports Node.js and comprises a bunch of innovations from previous versions, such as support for AngularJS, CoffeeScript and TypeScript, LESS, and SASS, and other great features.

This book is intended for you to learn about the newest features that WebStorm 10 provides and use WebStorm to exploit cutting-edge web technologies. You will see how to develop high-quality web applications and discover best practices and timesaving hacks in the web development process. Ultimately, you will gain all the skills required to revolutionize your web development experience.

In this chapter, you will do the following:

- Find out what is new in WebStorm 10
- Install and configure WebStorm
- Become familiar with the WebStorm workspace
- Set up and run a simple project
- Explore WebStorm settings and preferences

What is new in WebStorm 10?

WebStorm tends to improve and, thus, constantly acquires cutting-edge technologies, and Version 10 is quite demonstrative coming with an impressive arsenal of advanced features for full-scale web development:

- **Improved JavaScript support**: In this version of WebStorm, the support for JavaScript has been improved for larger projects with faster code highlighting and completion, enhanced ECMAScript 2015 support, and so on.
- **TypeScript**: This new version of WebStorm comes with support for versions 1.4 and 1.5 and a built-in compiler
- **Spy-js improvements**: WebStorm 10 adds the possibility of creating application dependency diagrams and tracing languages compiled to JavaScript
- **Grunt**: You can easily navigate and edit the jobs inside WebStorm. The grunt integration has been reworked to provide a consistent experience whether or not you decide to use Grunt or Gulp for your project
- **Live Dart analysis view**: You can now perform on-the-fly analysis for your code through the Dart Analysis Server. All the results will be displayed directly in the editor.

At the time of writing this book, JetBrains is already preparing to release WebStorm 11 that will bring cool features, such as Yeoman integration, advanced NPM integration, Webpack support, AngularJS 2 support, and improved JSX support to name a few.

Installing WebStorm

I believe that you are intrigued by all these features and are now longing to try out WebStorm 10 to leverage them. We need to install the IDE. You can find the download package directly on the WebStorm website at https://www.jetbrains.com/webstorm/.

One of the strongest advantages of WebStorm is that it's cross-platform. You can run WebStorm on the following operating systems:

- OS X 10.5 or higher, including 10.10
- Microsoft Windows 8/7/Vista/2003/XP (including 64-bit)
- GNU/Linux with GNOME or the KDE desktop

Besides, your machine should have the following configuration so as to painlessly run the IDE:

- A minimum of 512 MB of RAM (1 GB of RAM is recommended)
- Intel Pentium III/800 MHz or higher (or compatible)
- A minimum screen resolution of 1024 x 768
- Oracle (Sun) JDK 1.6+

Depending on the OS, the installation process varies slightly, but it still remains simple.

On a Mac machine, you should:

1. Mount the downloaded `WebStorm-10*.dmg` disk image file as another disk in your system.
2. Copy WebStorm to your `Applications` folder.

On a Windows machine, you should:

1. Run the downloaded `WebStorm-10*.exe` file that starts the installation wizard.
2. Follow all steps suggested by the wizard.

On a Linux machine, you should:

1. Unpack the downloaded `WebStorm-10*.tar.gz` file using the following command:

   ```
   tar xfz WebStorm-10*.tar.gz
   ```

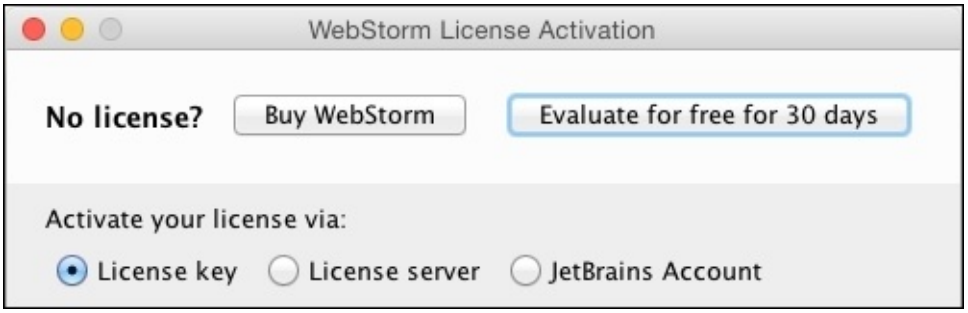
2. Move the extracted or unpacked archive folder to the desired location.
3. Run `WebStorm.sh` from the bin subdirectory.

Configuring WebStorm

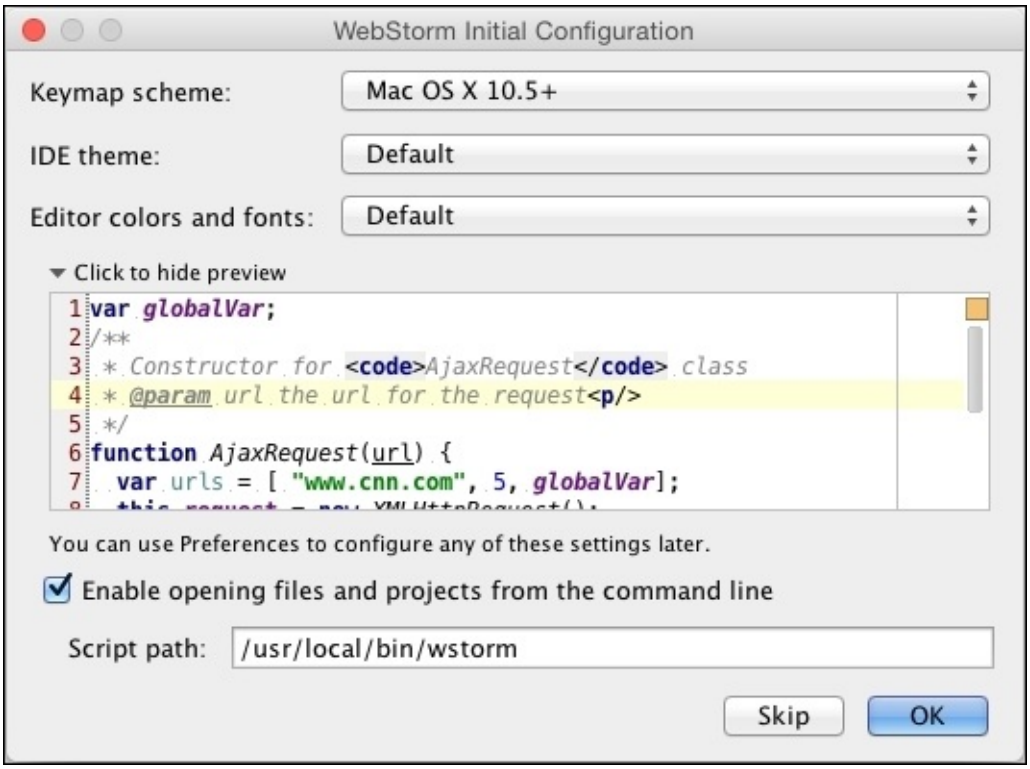
You need to configure a couple of things to complete the installation and proceed to work in the IDE.

First, it's suggested you import your settings from a previous version. However, since we assume here that you are new to WebStorm, click on the **I don't have a previous version of WebStorm or I don't want to import settings** button, as shown in the following screenshot:

Then, you get a popup of the **WebStorm License Activation** dialog, where you can activate your license if it exists, buy it, or run a 30-day WebStorm trial. Read the license agreement and accept all its terms. The following screenshot shows this:



When you are done with the license, in the **WebStorm Initial Configuration** dialog, you can set the **Keymap scheme** depending on what is habitual to you, the **IDE theme**, and the **Editor colors and fonts** depending on what you prefer more—dark or bright colors. Use the **Click to preview** section to evaluate whether the theme and colors you have set fit your needs and preferences or not. The shortcuts in this book are based on the **Mac OS X 10.5+** scheme. The following screenshot captures this discussion aptly:


WebStorm Initial Configuration
Keymap scheme:
Mac OS X 10.5+
IDE theme:
Default
Editor colors and fonts:
Default
Click to hide preview
1 var globalVar;
2 /**
3 * Constructor for <code>AjaxRequest</code> class
4 * @param url the url for the request<p/>
5 */
6 function AjaxRequest(url) {
7 var urls = ["www.cnn.com", 5, globalVar];
You can use Preferences to configure any of these settings later.
Enable opening files and projects from the command line
Script path:
/usr/local/bin/wstorm
Skip
OK

The user interface

Now we have WebStorm installed, so the time is ripe to look into its interface—the world where you are supposed to dwell when developing web applications. It is necessary that you understand the interface in order to be in your element and make your use in WebStorm efficient and pleasant.

Before you start

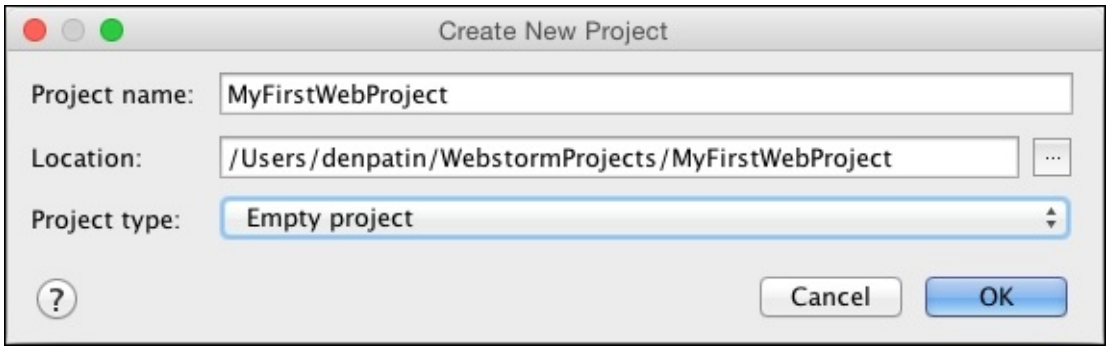
The first time that you run WebStorm or, later, when no project is open, you will see a welcome screen as follows:

You can either create a new project, open an existing one, check out the code from version control systems, or choose a setting action to perform from the **Configure** drop-down list. To get familiar with the WebStorm user interface, let’s create a new, simple HTML project.

Creating a new project

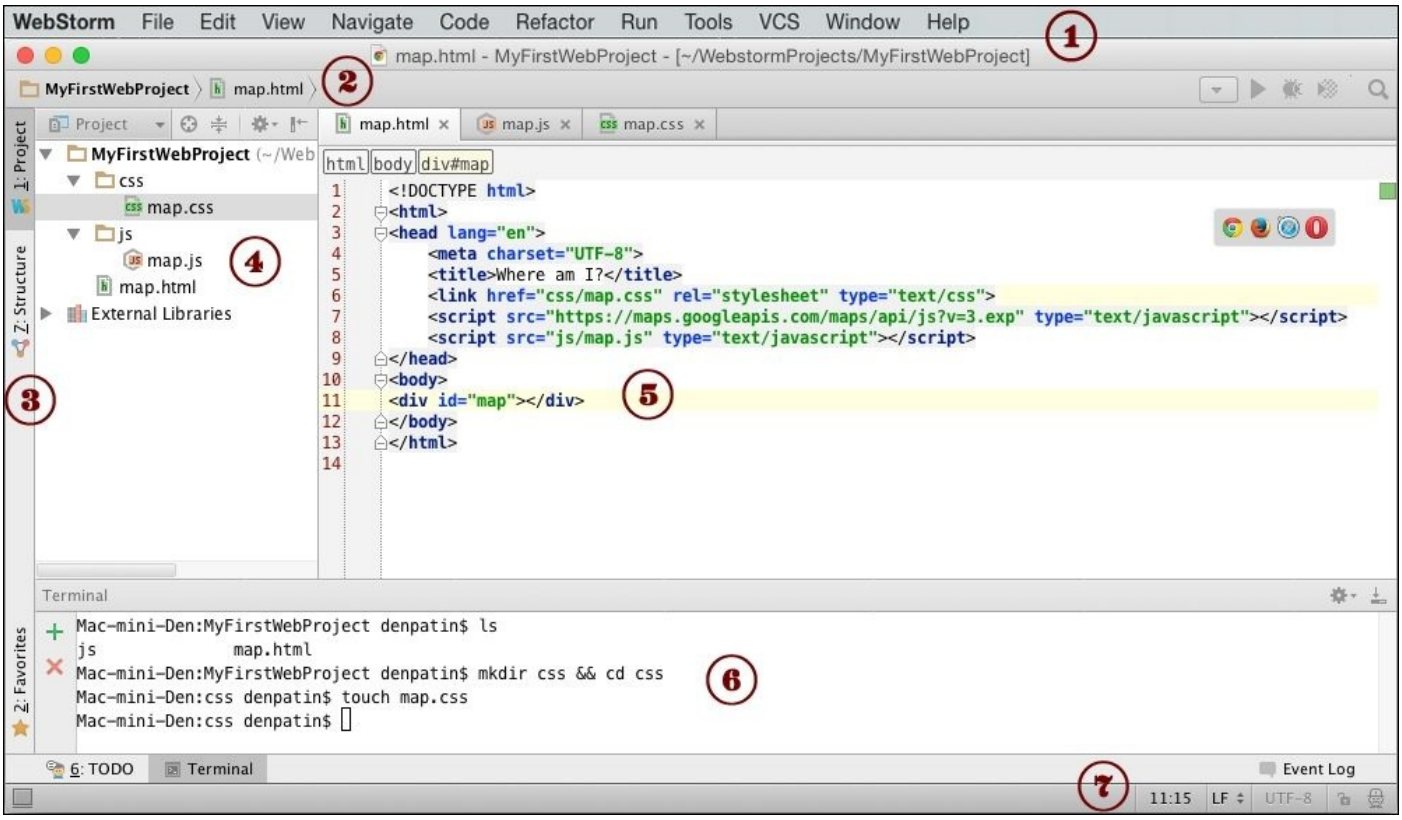
To create a new project, follow these steps:

1. Click on the **Create New Project** button, and the **New Project** dialog opens.
2. Give a name to our project, for example, `MyFirstWebProject`, and specify the folder for the project files, either manually or by choosing the folder using the browse (**…**) button.
3. Then, leave the **Project type** field with the **Empty project** value—for now, we are just going to immerse in the interface and not create something really useful—and click on the **OK** button. The following screenshot depicts these steps:



The WebStorm workspace

The WebStorm main window opens. It can be visually divided into seven logical parts, as shown in the following screenshot:



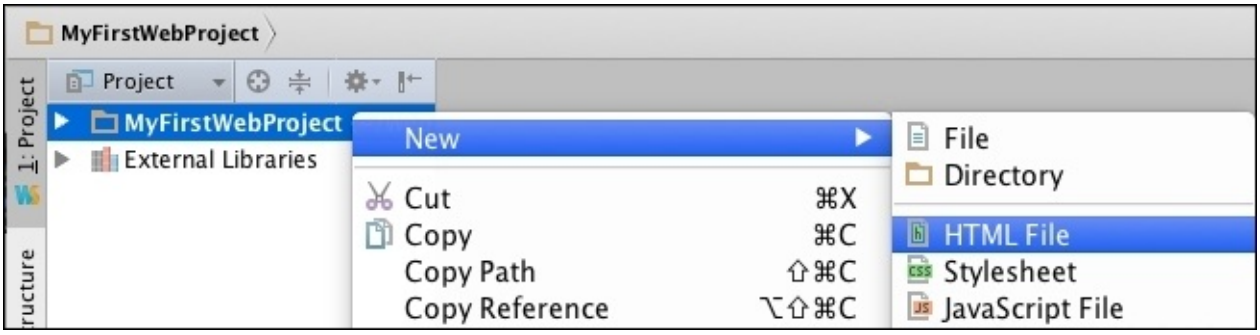
Instead of simply introducing them, let's create a very simple yet rather interesting project—that of automatically locating you with Google Maps. In the preceding image, you can see the final version of what we are going to do. So, after completing this section, you will have the same view.

So, what you can see now is the following:

- The **Menu line** (**1**), which has a bunch of options, including the standard **File**, **Edit**, **View** options and the WebStorm-specific **Code**, **Refactor**, **Run**, and **VCS** ones.
- The main window is divided into two plainly distinguishable parts:
    - The **Project tool panel** (**4**) on the left-hand side, where you will see your project structure and file. You now only have a line with the project name, yet in the image, you can already see the hierarchical structure of our project. The Project tool panel correlates with another interface element, **Tool tabs** (**3**), which I will describe a bit later.
    - The **Editor panel** (**5**) on the right-hand side, where you will type your code.
- Slightly above these two panels, you can find **Navigation toolbar** (**2**). For now, you have it with a sole breadcrumb, `MyFirstWebProject`, prepended with a typical folder icon and appended with a right-arrow shape, yet in the image, there are three breadcrumbs, where each one stands for the next hierarchy level. The Navigation

toolbar is great for quick navigation and better understanding of the project file hierarchy.
- At the very bottom, you can find **Status bar** (**7**), which contains some auxiliary information, such as denoting the current caret position in the editor (in the `line:column` format), line ending type (Windows-styled, Mac-styled, and so on), encoding the current file in the editor, and so on.

Your project is as yet empty. Let's get down to business! Add an HTML file to the project by right-clicking on the project name in the **Project** tool window, and name it `map.html` or simply `map`. This is encapsulated in the following screenshot:



Note

Note that unambiguously indicating the extension is optional because it is automatically added depending on the kind of field chosen.

In the editor, you see simple HTML5 starter code. There, let's indicate where our map will be displayed. It will be a simple `div` element with the `map` ID:

```
<div id="map"></div>
```

Note

I suggest you to keep from simply copying any code given in at least the first few chapters so that you acutely feel the advantages that WebStorm offers you.

In the `body` section, start typing `<div>`. As you type each symbol, what you see is that WebStorm suggests appropriate tags within the current content. You can navigate through the drop-down tag list with a mouse and select a tag with the *Enter* key in the same way that you can add attributes to the tag. When you type `>`, you will find that WebStorm automatically closes the tag if it is a pair tag. You should notice that WebStorm puts the cursor directly where you will further write the tag or attribute value; it slightly yet pleasantly saves some time. While you work on WebStorm, there is no need to save your files; by default, it will watch for changes in your files when you change your frame and save them automatically. This behavior can be set up/ modified in the **Preferences** | **Appearances & Behavior** | **System Settings** dialog

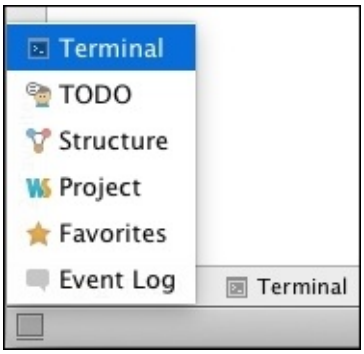
Now we need JavaScript code to render the map. To add it, first create a new directory, `js`, in the project root directory and add there a JavaScript file, `map.js`. It opens in a new tab and contains a comment on the author and the creation time. Fill the file with the

following code:

```
var map;

function initialize() {
    var mapOptions = {
        zoom: 10
    };
    map = new google.maps.Map(document.getElementById('map'), mapOptions);
    if(navigator.geolocation) {
        navigator.geolocation.getCurrentPosition(function(position) {
            var pos = new google.maps.LatLng(position.coords.latitude,
position.coords.longitude);
            var infowindow = new google.maps.InfoWindow({
                map: map,
                position: pos,
                content: 'I am here!'
            });
            map.setCenter(pos);
        }, function() {
            handleNoGeolocation(true);
        });
    } else {
        handleNoGeolocation(false);
    }
}

function handleNoGeolocation(errorFlag) {
    if (errorFlag) {
        var content = 'Error: The Geolocation service failed.';
    } else {
        var content = 'Error: Your browser doesn\'t support geolocation.';
    }
    var options = {
        map: map,
        position: new google.maps.LatLng(60, 105),
        content: content
    };
    var infowindow = new google.maps.InfoWindow(options);
    map.setCenter(options.position);
}

google.maps.event.addDomListener(window, 'load', initialize);
```

The code we just created uses the Google Maps API's features, renders the map on your HTML page, locates you, and shows an **I am here!** message there.

Let's now pay attention to one more workspace element. It is located at almost the bottom of the IDE (6). Doesn't this remind you of something? Yes, it is a standard Terminal of your OS, which is integrated into WebStorm. You can activate it in the bottom-left corner by either clicking on the **Terminal** button, or selecting the **Terminal** option in the drop-down option list when clicking on the icon, as shown in the following screenshot:



The terminal opens, and you can use it as if you were using the terminal built in to your OS. I want you to create one more directory—`css`—to keep the `map.css` file via the WebStorm Terminal. Simply execute the following commands, and both the `css` directory and `map.css` file appear in the **Project** tool panel:

```
$ mkdir css
$ cd css
```

You will now have a new directory named `css`. In this folder, create a new CSS file named `map.css` from the context dialog or **File** | **New** menu. There is no advantage in using the terminal in preference to context menu manipulations, or vice versa. Which you use is up to you, but since we are exploring an instrument, WebStorm, that is new to you, I will nevertheless prefer using its interface as much as possible.

Now double-click on the `map.css` file in the project structure hierarchy and fill it with the following code:

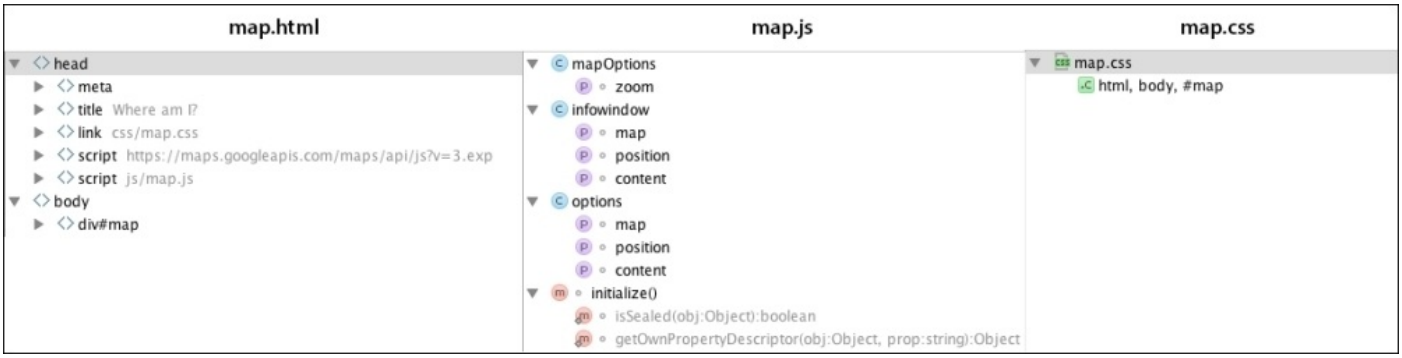
```
html, body, #map {
    height: 100%;
    margin: 0;
    padding: 0
}
```

Now we need to add the finishing touch—we should tell HTML to use our CSS and JavaScript files as well as point out where the Google Maps API is located. We can do it by simply adding the link and script tags to the `head` section. In the same way that it was described, you can use WebStorm's code completion feature to speed up your code typing. I want to mention one more feature now which is really essential here (we will talk about WebStorm's smart code completion in detail in the next chapter). It concerns those attributes that point to the other project files—in our case, it is the `src` and `href` attributes. There is no need to manually search for files to point at from these tags (it is especially appreciable in huge projects)—WebStorm does everything for you. When already being between quotes and waiting to input an attribute value, you can simply press on *Ctrl* + *Space* (on Windows) / *Ctrl* + *Space* (on Mac), and WebStorm displays only those files in the drop-down list that are possible in this context. If there are too many files, you can simply start typing the name of the file and WebStorm's smart completion does its work, too. Of course, it is a scintilla of what WebStorm can offer you. You will become familiar with a bundle of smart techniques in the next chapter.

So, the resulting HTML code is the following:

```
<!DOCTYPE html>
<html>
<head lang="en">
    <meta charset="UTF-8">
    <title>Where am I?</title>
    <link href="css/map.css" rel="stylesheet" type="text/css">
    <script src="https://maps.googleapis.com/maps/api/js?v=3.exp"
type="text/javascript"></script>
    <script src="js/map.js" type="text/javascript"></script>
</head>
<body>
<div id="map"></div>
</body>
</html>
```

The last yet not least important interface element is Tool tabs (3), which I have already mentioned, but only now, we can see the benefits of using it. Earlier, we were on the **Project** tab, where we could see the file and directory hierarchy. Change the tab for the **Structure** one, and for each file—`map.html`, `map.js`, and `map.css`—you will see the following:



This is also a hierarchy, but not of filesystem elements; this is a content-based hierarchy. For HTML, it means tag hierarchy; for CSS, all styles are enumerated; and for JavaScript, you can see information about all objects and functions with parameters. Moreover, clicking on any line moves you directly to the place in the code where the element on this line is declared or exists.

Voilà, your first project within WebStorm is ready! Now it’s time to run our page.

Running the application

Do you remember that the purpose of any IDE is to take as many actions as possible upon itself in order to help you to only concentrate on designing projects and typing code, instead of performing a huge number of minor yet numerous actions? It means that most of the time, you will stay within the four walls of the IDE, performing almost all actions from inside it. If you need to perform something outside the IDE, give a thought to whether you can perform it by means of the IDE. However, there is one thing for which you will need to leave the IDE—it is your browser. You need to see your results, don't you? But even in this case, WebStorm helps you. When you were typing the HTML code, did you notice a browser icon list panel in the top-right corner in the editor? Here's what the panel looks like:

Yes, each icon is a shortcut to a browser executable, and if you click on any one, the corresponding browser opens with the current page. Note that only installed browsers will open.

The page with your current location opens at the address `http://localhost:63342/MyFirstWebProject/map.html`. Don't be afraid if you see a blank page. First, you need to allow your browser to get your location as it will ask about it. Then, you will see a map wherein your location will be indicated with the words **I am here!**.

The only question that remains is, what is the strange number after the `localhost` word? WebStorm provides the facility of a simple HTTP server running on the default port `63342`. You can change the port in IDE Settings.

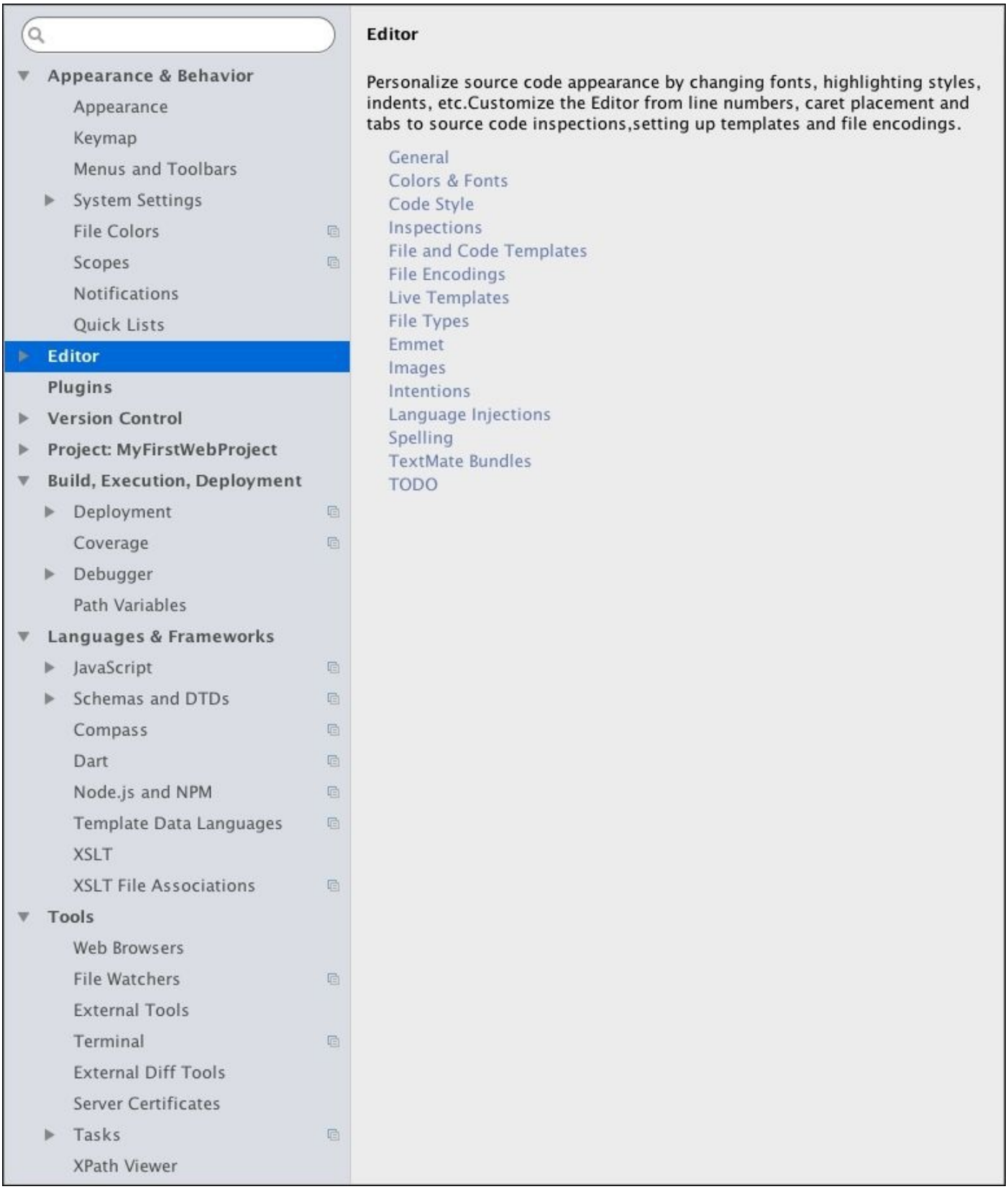
Settings and preferences

By now, we are going to speak about customizing something not in the code, but in WebStorm itself, so I must tell you about the multifarious settings that WebStorm offers.

There are two similar yet separate settings windows:

- The settings of the whole IDE
- The settings of the current project

For the first case, to open the settings window, you can simply press either *Ctrl* + *Alt* + *S* (Windows) or ⌘ + , (Mac), and you will get the following screen:



To access project settings, select **File | Default Settings**, and you will get something similar to the WebStorm settings, but with a reduced list of configurable options.

Note

I should say at once that settings are not a thing that matters every day, so you needn't think of it as something to necessarily be utterly and completely configured—in most

cases, everything is already tuned up; moreover, if something needs configuring, it will pop up as an information balloon with a direct link to a setting.

If you need to configure something, what you will need to do is quickly find the necessary setting. To do that, start typing its name in the search field, and, as you type, the first matching setting gets highlighted, and the corresponding page is displayed.

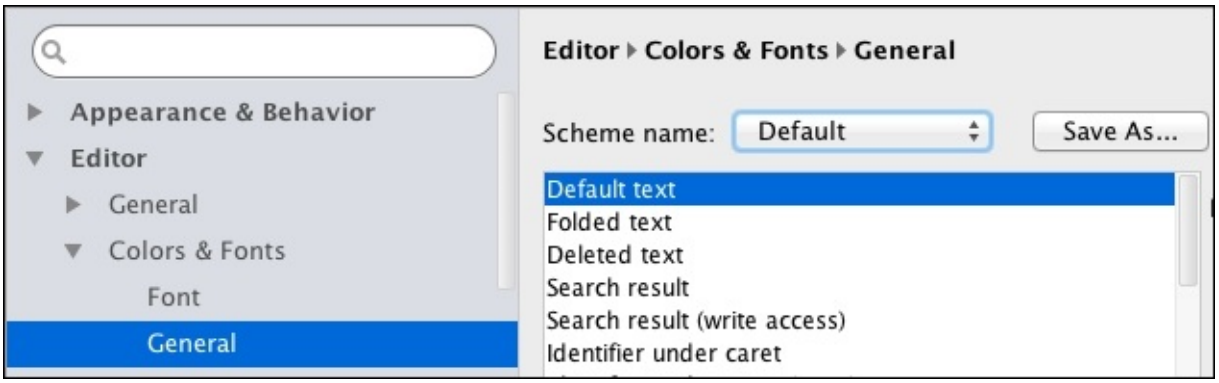
But let’s stroll further through the settings. I am not aiming to describe all the settings, but still I am going to cover the most important ones.

Themes and colors

At the stage of initial configuration, you chose a theme and the colors you wanted your IDE to have. Maybe during the process of programming, you realized that you'd better use another theme and/or color.

If you want to change a theme of the IDE, go to the **Appearance** subsection of the **Appearance & Behavior** section. There, you will find the **Theme** drop-down list where you can change your current theme, as shown in the following screenshot:

If you want to change a color for a certain section of your IDE, you can go to the **Colors & Fonts** subsection of the **Editor** section and try various parameters there so as to find the ones that suit you. You can observe the changes in real time by clicking on the **Apply** button. This discussion is aptly depicted in the following screenshot:



Keymap

The keymap is that very thing that developers must know and feel on the tips of their fingers because the proficiency level to use shortcut keys is directly proportional to the speed of development. It is evident that each person has his own preferences of what key combination to use in each case. So, the IDE should take into account the necessity to customize the keymap so as to suit a larger number of developers.

WebStorm offers you comprehensive keymap management, which you can access in the **Keymap** subsection of the **Appearance & Behavior** section. For each menu item and command accessible inside the IDE, you get an opportunity to change the default key combination. Of course, WebStorm prevents you from overwriting and overriding the already existing combinations with a warning message.

To add a new shortcut or to change an existing one, you need to just select what you are going to process and click on the **Remove** or/and **Add Keyboard Shortcut** button—depending on what you intend to do. Then, simply press the combination that you would like to set for a certain action, and that’s all.

This is great news for those who migrated from one OS to another or from one IDE to WebStorm. If you are already used to a keymap specific to your previous workspace, you need to first check the **Keymap** options list, where you can reliably find an option that satisfies you. Among the options are Emacs, Visual Studio, Eclipse, NetBeans, Mac OS X, Linux, Windows, and so on.

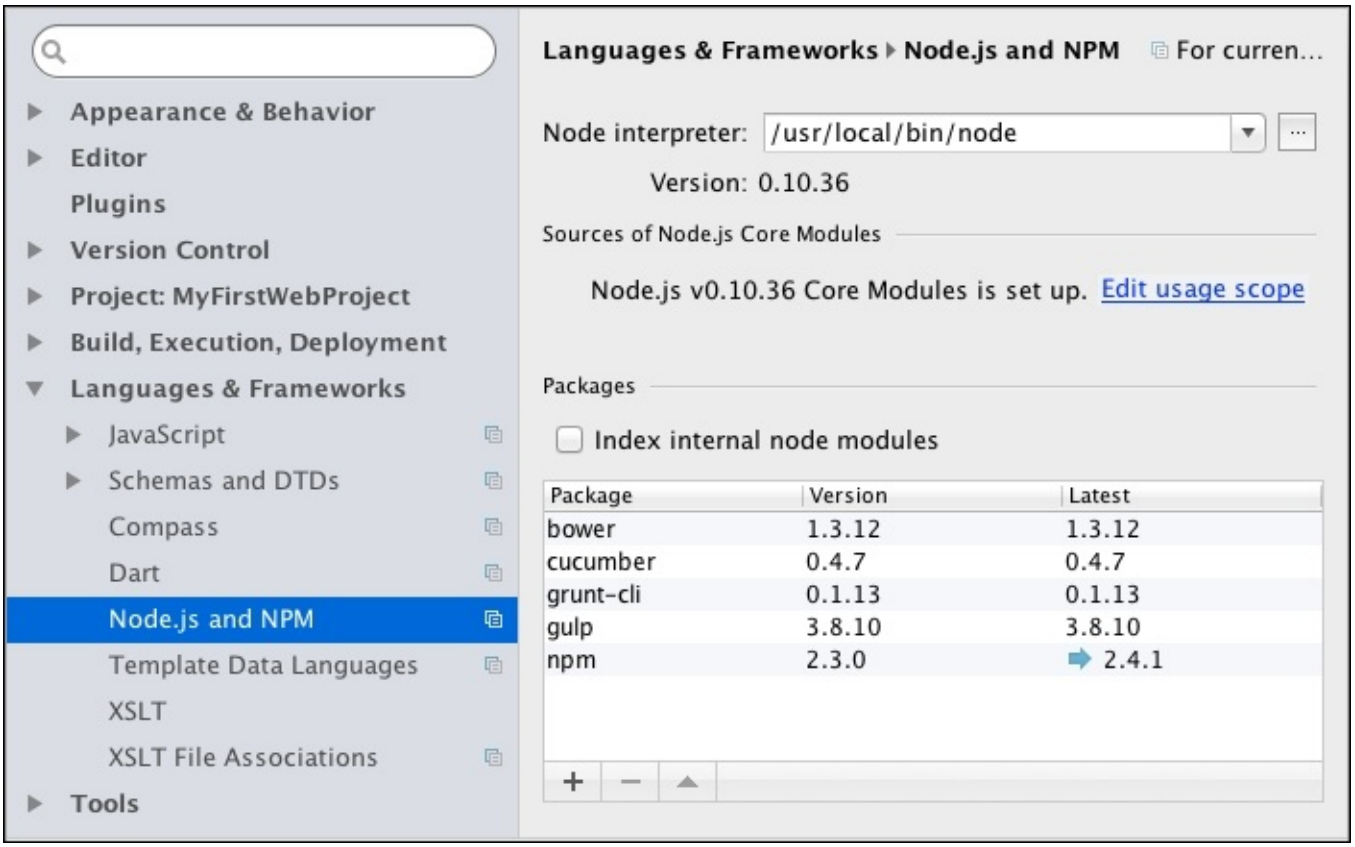
Code Style

In the **Code Style** subsection of the **Editor** section, you can see the possibility of customizing the appearance of each language in which you an write the code using WebStorm: JavaScript, CoffeeScript, CSS, Haml, JSON, Dart, XML, YAML, and so on.

Settings about the languages that you can change are uncountable. You can set the indent size, indent type, and if you are using tabs, their size, where to put or not put spaces, how to perform wrapping, and lot of other aspects. On the customization panel for each language is a preview of what you get if you change something.

Languages & Frameworks

There is a **Languages & Frameworks** section in settings that is responsible for the technologies considered to be used. Here, you can specify the executables for the necessary frameworks and language interpreters, set the default parameters to run with, add packages for them, and so on. The following screenshot encapsulates the discussion aptly:



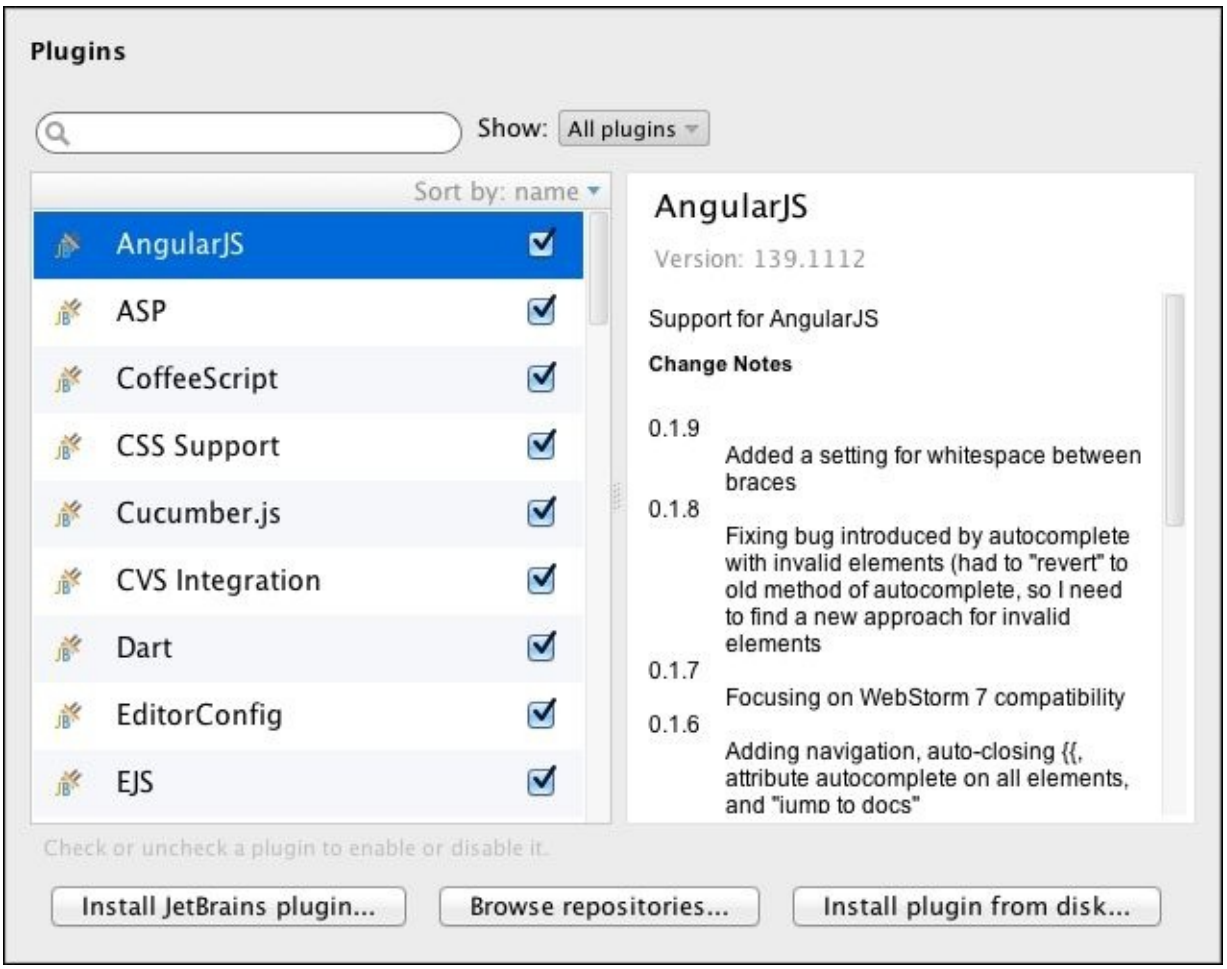
Plugins

Plugins are an important part of WebStorm because they can dramatically expand the core functionality of the IDE.

Note

Be selective and do not install the plugins one by one. Remember that the more plugins you install, the more the time it will take you to launch the IDE, and the poorer the performance it will have. But don’t be afraid of installing plugins—each one adds just a couple of dozens of milliseconds to the startup time of your IDE, so you will not experience performance degradation unless you install only necessary plugins. Up to 10-15 plugins is fine for performance as well as enough for a typical project.

You can manage plugins in the **Plugins** section of the **Preferences** window, as shown in the following screenshot:



The plugins that are checked are already installed and accessible inside the IDE. If you need more plugins, there are two ways of getting them. You can either browse them in online repositories or install already downloaded or engineered plugins directly from the disk.

The difference between the **Install JetBrains plugin…** window and the **Browse repositories…** window is that the second one allows you to install any plugin available in

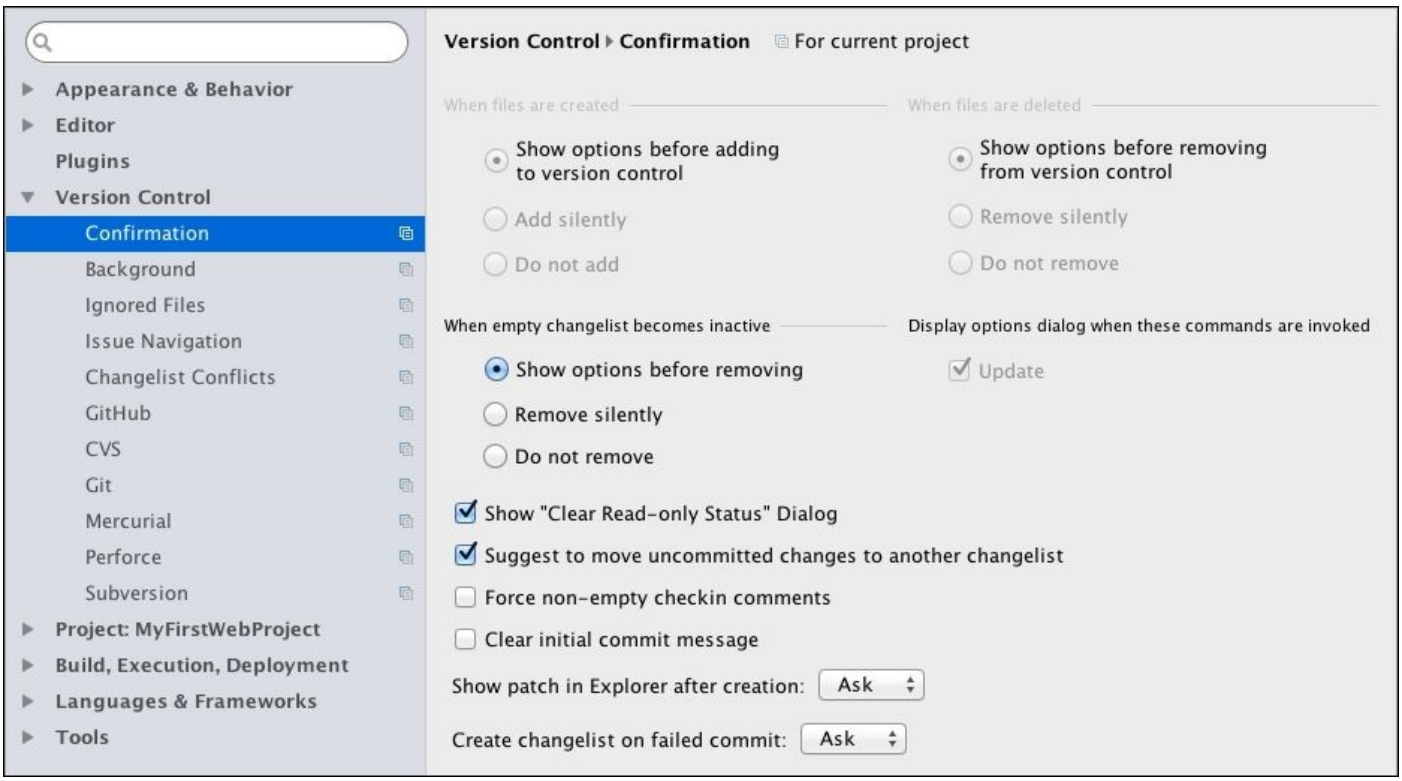
online repositories, whether it be third-party or of JetBrains.

There is nothing about it in the installation process of a plugin. You need to just find what you want and simply click on the **Install Plugin from disk…** button.

Version Control

Any development process, be it web, desktop, or whatever else, is inconceivable without version control. That commits IDEs to integrate various version control systems, and WebStorm is not an exception. WebStorm supports **Git**, **Subversion**, **Mercurial**, **CVS**, and **Perforce** besides providing a binding with your GitHub account. Besides, WebStorm provides you with the facility of the so-called “local history”, which enables you to track any changes in the code by means of the IDE, without using dedicated VCSes. Despite the differences between these VCSes, WebStorm allows you to use them in similar ways. As an IDE, WebStorm, of course, takes a lot of issues, so dealing with VCSes is visual and becomes even simpler than via the standard command line.

You can decide which actions require confirmation (in the following image), specify which ones can run in the background, configure what is to be ignored, manage history cache handling, and so on. Then, you will be able to unnoticeably yet skillfully manage your VCS using just shortcut keys. **Version Control** is depicted in the following screenshot:

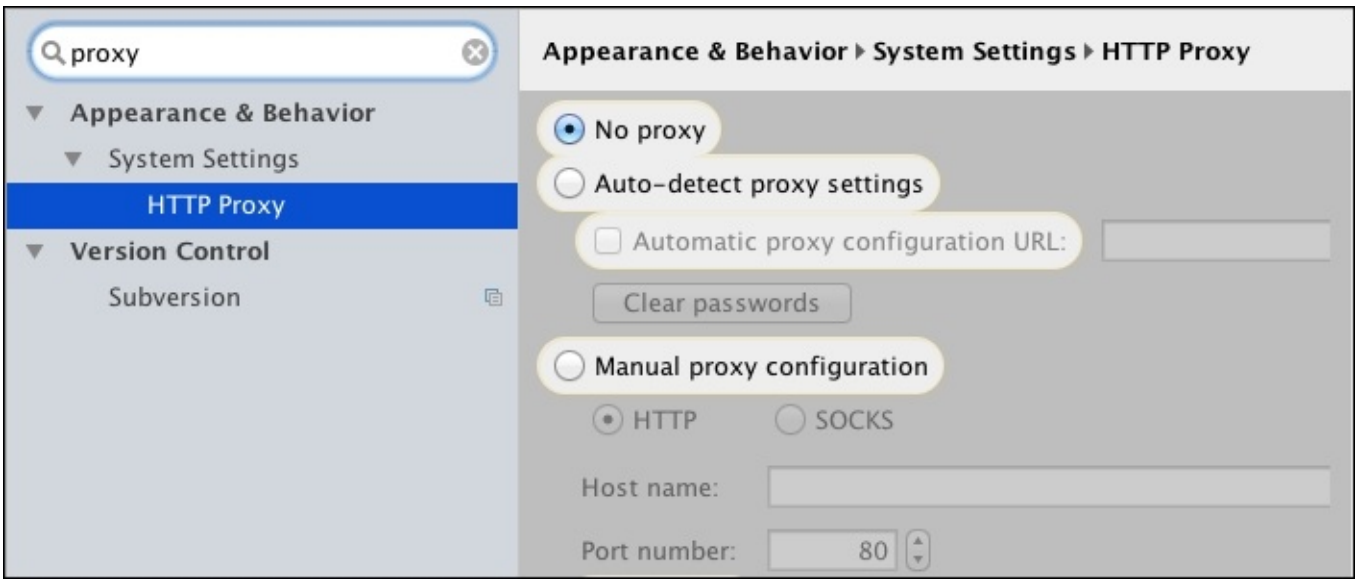


Proxy

You may need to use WebStorm at work. A lot of companies are using internal networks, so it may turn out in a way that you will not be able to perform most Internet-oriented actions without a proxy. There may also be something else, which prevents you from comfortably working on the Internet. So, let’s learn how to set up a proxy in WebStorm.

Instead of manually searching where the proxy settings are located, let’s query the search box about it, entering the word `proxy` there. It will instantly not only show where the proxy settings are, but also filter out only the menu sections containing “proxy”. Moreover, it highlights all the parameters that contain the word “proxy”, so we can rapidly find what we are searching for.

For the proxy to process connections is a small matter. You need to fill out the blank fields as displayed here and click on the **Apply** button:



Summary

In this chapter, you learned how to install and then prepare WebStorm for the development process. We implemented a simple web application, and by the example of the files that we created and the actions we performed, we scrutinized the WebStorm interface and the means to customize the most significant elements of it.

In the next chapter, we are going to discover a bunch of the most essential and smart features that drastically improve your efficiency and, thus, makes WebStorm really great for web development.

Chapter 2. Improving Your Efficiency with Smart Features

In the previous chapter, you embarked on the path of familiarizing yourself with the world of WebStorm. You have not just learned about a couple of basic things, but also created a simple project using several features so you now have the first-hand elementary experience of using WebStorm, and now it is time to go further and cultivate your skills.

In this chapter, we are going to deal with a number of really smart features that will enable you to fundamentally change your approach to web development and learn how to gain maximum benefit from WebStorm. We are going to study the following in this chapter:

- Syntax highlighting
- On-the-fly code analysis
- Smart code features
- Multiselect feature
- Refactoring facility
- Advanced navigation

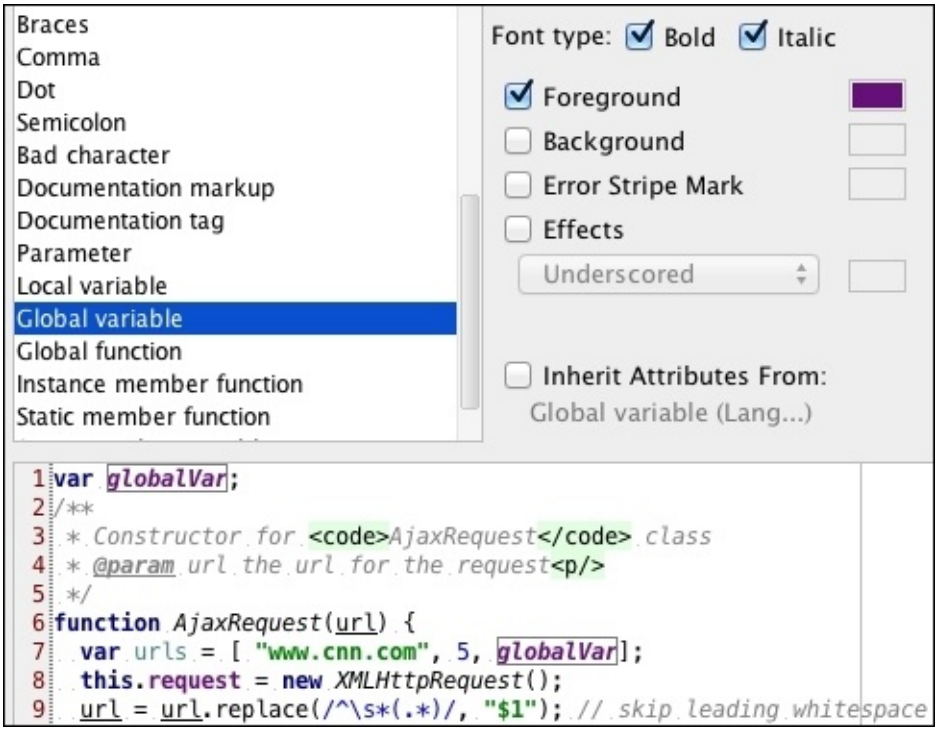
Syntax highlighting

Syntax highlighting facility is one of the core things that any code editor must be great at, because even a dozen of uncolored lines of code may be confusing and even irritating. Further, you can see how WebStorm performs the highlighting:

```
defaultOptions.items.push({
  key: 'commentsRemove',
  name: function () {
    return "Delete Comment"
  },
  callback: function (key, selection, event) {
    Handsontable.Comments.removeComment(selection.start.row, selection.start.col);
  },
  disabled: function () {
    var hasComment = Handsontable.Comments.checkSelectionCommentsConsistency();
```

WebStorm not only neatly highlights the code, but also provides you with a bunch of flexible settings for doing this. It means that you are able to customize each particle of the code appearance completely up to your preferences. You are enabled to customize highlighting for JavaScript and CoffeeScript and TypeScript, CSS and SASS/SCSS, JSON and XML and YAML, and so on.

To configure syntax highlighting, open the **Preferences** window, then go to the **Colors & Fonts** subsection of the **Editor** section, and choose the language or technology for customizing its code styling:



For each code unit—keywords, tags, numbers and words, comments, punctuation signs, variable and function declarations—you can change its appearance—bold or italic, the

color of the foreground and background, font effects, and much more. Below the configuration panel, you can find a preview panel where you can observe all the changes you performed in real time. Besides, if you know what code unit you would like to customize but don’t know what it is called, you can simply click on any sign or word of the code in the preview panel, and it will be automatically chosen in the list of code unit names, as well as all occurrences of the same type of unit get to blink so you can see what you are choosing.

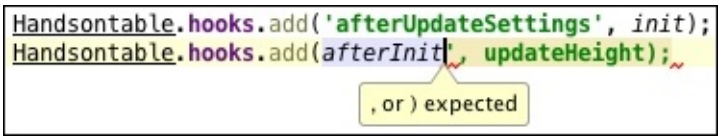
After customizing, you can save all settings as a new scheme so that the settings can be applied for further projects, as well as this one. To do this, you can simply click the **Save As…** button above the settings, and name this new code style scheme.

WebStorm is also able to detect the language you are using in specific contexts and will highlight the code accordingly. So, for example, if you are writing JavaScript code inside an HTML file, WebStorm will be able to detect this and highlight your code correctly.

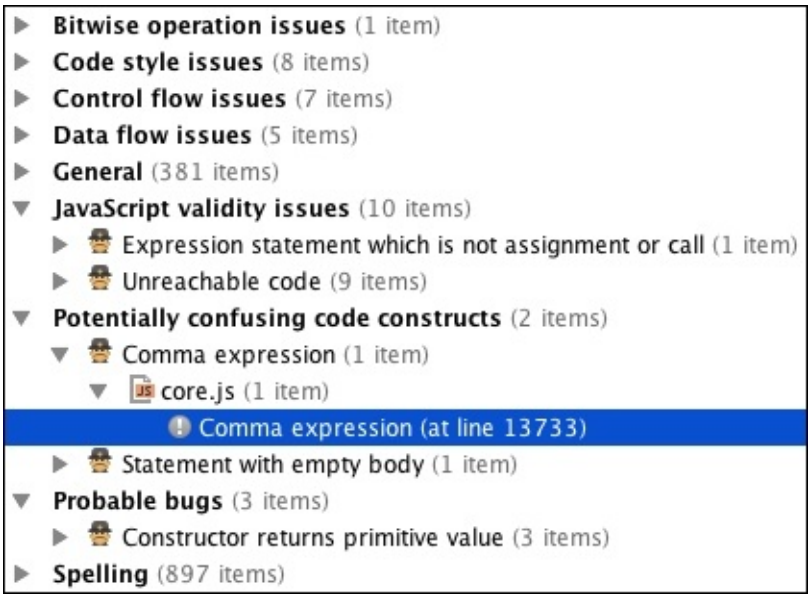
On-the-fly code analysis

WebStorm will perform static code analysis on your code on the fly. The editor will check the code, based on the language used, and the rules you specify and highlight warnings and errors as you type. This is a very powerful feature that means you don’t need to have an external linter and will catch most errors quickly, thus making a dynamic and complex language like JavaScript more predictable and easy to use.

Runtime error and any other error, such as syntax or performance, are two things to consider. To investigate the first one, you need tests or a debugger, and it is obvious that they have almost nothing in common with the IDE itself (although, when these facilities are integrated into the IDE, such a synergy is better, but that is not it). You can also examine the second type of errors the same way, but is it convenient? Just imagine that you need to run tests after writing the next line of code. It is no go! Won’t it be more efficient and helpful to use something that keeps an eye on and analyzes each word being typed in order to notify you about probable performance issues and bugs, code style and workflow issues, various validation issues, warn of dead code and other likely execution issues before executing the code, to say nothing of reporting inadvertent misprints. WebStorm is the best fit for it. It performs a deep-level analysis of each line, each word in the code. Moreover, you needn’t break off your developing process when WebStorm scans your code; it is performed on the fly and thus so called:

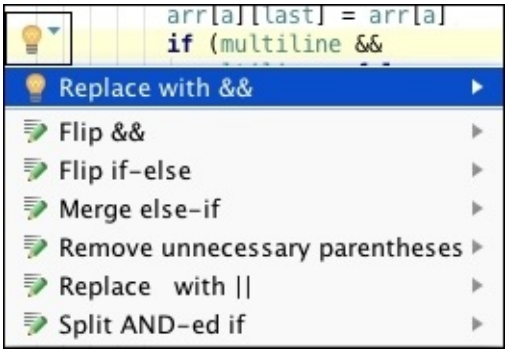


WebStorm also enables you to get a full inspection report on demand. To get it, go to the menu: **Code** | **Inspect Code**. It pops up the **Specify Inspection Scope** dialog where you can define what exactly you would like to inspect, and click **OK**. Depending on what is selected and of what size, you need to wait a little for the process to finish, and you will see the detailed results where the Terminal window is located:



You can expand all the items, if needed. To the right of this inspection result list you can see an explanation window. To jump to the erroneous code lines, you can simply click on the necessary item, and you will flip into the corresponding line.

Besides simple indicating where some issue is located, WebStorm also unequivocally suggests the ways to eliminate this issue. And you needn't even make any changes yourself—WebStorm already has quick solutions, which you need just to click on, and they will be instantly inserted into the code:



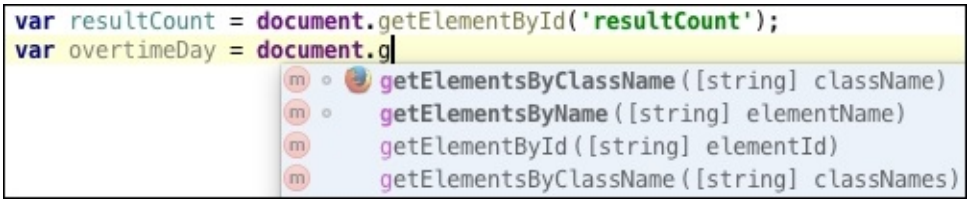
We will go more into the details of code analysis and quality in Chapter 7, *Analyzing and Debugging Your Code*.

Smart code features

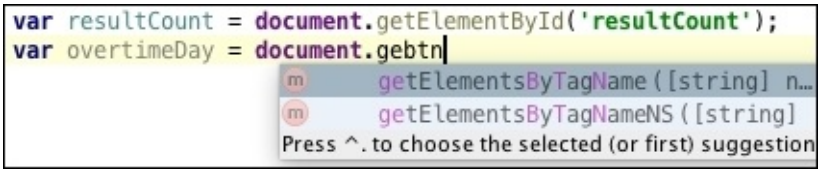
Being an **Integrated Development Environment** (**IDE**) and tending to be intelligent, WebStorm provides a really powerful pack of features which you can use to strongly improve your efficiency and save a lot of time.

One of the most useful and hot features is code completion. WebStorm continually analyzes and processes the code of the whole project, and smartly suggests the pieces of code appropriate in the current context, and even more—alongside the method names you can find the usage of these methods. Of course, code completion itself is not a fresh innovation, but WebStorm performs it in a much smarter way than other IDEs do. WebStorm can auto-complete a lot things: Class and function names, keywords and parameters, types and properties, punctuation, and even file paths.

By default, the code completion facility is on. To invoke it, simply start typing some code. For example, in the following image you can see how WebStorm suggests object methods:



You can navigate through the list of suggestions using your mouse or the *Up* and *Down* arrow keys. However, the list can be very long, which makes it not very convenient to browse. To reduce it and retain only the things appropriate in the current context, keep on typing the next letters. Besides typing only initial consecutive letter of the method, you can either type something from the middle of the method name, or even use the CamelCase style, which is usually the quickest way of typing really long method names:

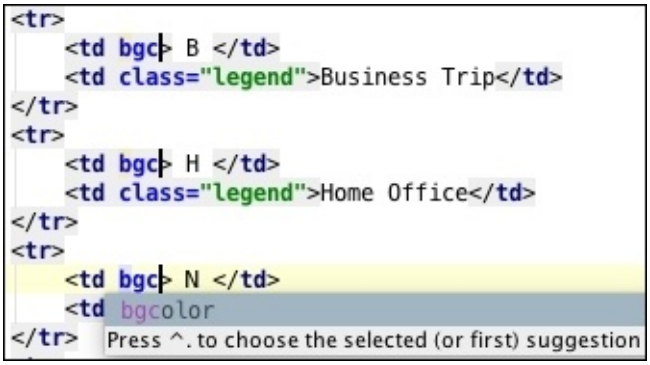


It may turn out for some reason that the code completion isn’t working automatically. To manually invoke it, press *Ctrl* + *Space* on Mac or *Ctrl* + *Space* on Windows.

To insert the suggested method, press *Enter*; to replace the string next to the current cursor position with the suggested method, press *Tab*. If you want the facility to also arrange correct syntactic surroundings for the method, press *Shift* + ⌘ + *Enter* on Mac or *Ctrl* + *Shift* + *Enter* on Windows, and missing brackets and/or new lines will be inserted, up to the styling standards of the current language of the code.

The multiselect feature

With the multiple selection (or simply multiselect) feature, you can place the cursor in several locations simultaneously, and when you will type the code it will be applied at all these positions. For example, you need to add different background colors for each table cell, and then make them of twenty-pixel width. In this case, you don’t need to perform these identical tasks repeatedly and can save a lot of time by placing the cursor after the `<td>` tag, press *Alt*, and put the cursor in each `<td>` tag, which you are going to apply styling to:



Now you can start typing the necessary attribute—it is `bgcolor`. Note that WebStorm performs smart code completion here too, independently of you typing something on a single line or not. You get empty values for `bgcolor` attributes, and you fill them out individually a bit later. You need also to change the width so you can continue typing. As cell widths are arranged to be fixed-sized, simply add the value for `width` attributes as well. An example is shown in the following image:

Moreover, the multiselect feature can select identical values or just words independently, that is, you needn’t place the cursor in multiple locations. Let us look at this feature by using another example. Say, you changed your mind and decided to colorize not backgrounds but borders of several consecutive cells. You may instantly think of using a simple replace feature but you needn’t replace all attribute occurrences, only several consecutive ones. For doing this, you can place the cursor on the first attribute, which you are going to perform changes from, and click *Ctrl + G* on Mac or *Alt + J* on Windows as many times as you need. One by one the same attributes will be selected, and you can

replace the `bgcolor` attribute for the `bordercolor` one:

```
<tr>
    <td bordercolor="#DE6FAC" width="20"> B </td>
    <td class="legend">Business Trip</td>
</tr>
<tr>
    <td bordercolor="#FFFFFF" width="20"> H </td>
    <td class="legend">Home Office</td>
</tr>
<tr>
    <td bordercolor="#DDDDDD" width="20"> N </td>
    <td class="legend">NA</td>
```

You can also select all occurrences of any word by clicking *Ctrl + command + G* on Mac or *Ctrl + Alt + Shift + J*.

To get out of the multiselect mode you have to click in a different position or use the *Esc* key.

Refactoring facility

Throughout the development process, it is almost unavoidable that you have to use refactoring. Also, the bigger code base you have, the more difficult it becomes to control the code, and when you need to refactor some code, you will most likely be up against some issues relating to, for example, naming omission or not taking into consideration function usage. You learned that WebStorm performs a thorough code analysis so it understands what is connected with what and, if some changes occur, it collates them and decides what is acceptable, and what is not to perform in the rest of the code. Let us try a simple example.

In a big HTML file you have the following line:

```
<input id="search" type="search" placeholder="search" />
```

And in a big JavaScript file you have another one:

```
var search = document.getElementById('search');
```

You decided to rename the `id` attribute's value of the input element to `search_field` because it is less confusing. You could simply rename it here but after that you would have to manually find all the occurrences of the word `search` in the code. It is evident that the word is rather frequent so you would spend a lot of time recognizing usage cases appropriate in the current context or not. And there is a high probability that you would forget something important, and even more time will be spent on investigating an issue. Instead, you can entrust WebStorm with this task. Select the code unit to refactor (in our case, it is the `search` value of the `id` attribute), and click *Ctrl + T* on Mac or *Ctrl + Alt + Shift + T* on Windows (or simply click the **Refactor** menu item) to call the **Refactor This** dialog. There, choose the **Rename…** item and enter the new name for the selected code unit (`search_field` in our case). To get only a preview of what will happen during the refactoring process, click the **Preview** button, and all the changes to apply will be displayed in the bottom. You can walk through the hierarchical tree and either apply the change by clicking the **Do Refactor** button, or not. If you need a preview, you can simply click the **Refactor** button. What you will see is that the `id` attribute got the `search_field` value, not the type or placeholder values, even if they have the same value, and in the JavaScript file you got `getElementById('search_field')`.

Note

Note that even though WebStorm can perform various smart tasks, it still remains a program, and some issues can be caused by so-called artificial intelligence imperfection, so you should always be careful when performing the refactoring. In particular, manually check the `var` declarations because WebStorm sometimes can apply the changes to them as well, but it is not always necessary because of the scope.

Of course, this is just a little of what you are enabled to perform with refactoring. The basic things that the refactoring facility allows you to do are as follows:

The elements in the preceding screenshot are explained as follows:

- **Rename…**: You have already got familiar with this refactoring. Once again, with it you can rename code units, and WebStorm automatically will fix all references of them in the code. The shortcut is *Shift* + *F6*.
- **Change Signature…**: This feature is used for changing function names, and adding/removing, reordering, or renaming function parameters, that is, changing the function signature. The shortcut is ⌘ + *F6* for Mac and *Ctrl* + *F6* for Windows.
- **Move…**: This feature enables you to move files or directories within a project, and it simultaneously repairs all references to these project elements in the code so you needn't manually repair them. The shortcut is *F6*.
- **Copy…**: With this feature, you can copy a file or directory or even a class, with its structure, from one place to another. The shortcut is *F5*.
- **Safe Delete…**: This feature is really helpful. It allows you to safely delete any code or entire files from the project. When performing this refactoring, you will be asked about whether it is necessary to inspect comments and strings or all text files for the occurrence of the required piece of code or not. The shortcut is ⌘ + *delete* for Mac and *Alt* + *Delete* for Windows.
- **Variable…**: This refactoring feature declares a new variable where the result of the selected statement or expression is put. It can be useful when you realize there are too many occurrences of a certain expression so it can be turned into a variable, and the expression can just initialize it. The shortcut is *Alt* +⌘ + *V* for Mac and *Ctrl* + *Alt* + *V* for Windows.
- **Parameter…**: When you need to add a new parameter to some method and appropriately update its calls, use this feature. The shortcut is *Alt* + ⌘ + *P* for Mac and *Ctrl* + *Alt* + *P* for Windows.
- **Method…**: During this refactoring, the code block you selected undergoes analysis, through which the input and output variables get detected, and the extracted function receives the output variable as a return value. The shortcut is *Alt* + ⌘ + *M* for Mac and *Ctrl* + *Alt* + *M* for Windows.
- **Inline…**: The inline refactoring works contrariwise to the extract method refactoring —it replaces surplus variables with their initializers making the code more compact and concise. The shortcut is *Alt* + ⌘ + *N* for Mac and *Ctrl* + *Alt* + *N* for Windows.

Advanced navigation

Navigation is one of the most important things in any IDE because if you cannot quickly find or switch over to whatever you want, there is little reason to use such a program, as there is no time saving aspect.

Partially, we examined some navigation features WebStorm provides. They are the menu panel, the navigation toolbar, and tool tabs. It is all great but we still need to perform a lot of mouse movements to reach the necessary place in the code. We are going to see how WebStorm can help us working with code by going through some of these power navigation tools. We are going to go through them based on what type of action they can help us perform.

File navigations

This are a set of shortcuts that help us work efficiently with files:

- To navigate to a file in the project, press *Shift* + ⌘ + *O* on Mac or *Ctrl* + *Shift* + *N* on Windows.
- To navigate between the files opened in the editor, press *Ctrl* + *Tab* on Mac or *Alt* + *Tab* on Windows.
- To see what files were last opened, press ⌘ + *E* on Mac or *Ctrl* + *E* on Windows. To see what files were recently updated, press *Shift* + ⌘ + *E* on Mac or *Ctrl* + *Shift* + *E* on Windows.
- To navigate to a class in the project, press ⌘ + *O* on Mac or *Ctrl* + *N* on Windows.

Code navigations

These shortcuts are designed to help us to navigate and quickly perform tasks on our source code:

- To pop up a structure view of the file, press ⌘ + *F12* on Mac or *Ctrl* + *F12* on Windows.
- To get hierarchy for the selected class, press *Ctrl* + *H* on Mac or *Ctrl* + *H* on Windows.
- To navigate to a declaration in the project, press ⌘ + *B* on Mac or *Ctrl* + *B* on Windows.
- To navigate to the next method, press ⌘ + *O* on Mac or *Alt* + *Down* arrow key on Windows. To navigate to the previous method, press ⌘ + *O* on Mac or *Alt* + *Up* arrow key on Windows.
- If you need to duplicate the current line, press *Ctrl* + *D* on Mac or *Alt* + *D* on Windows.
- If you need to comment/uncomment the line, press ⌘ + / on Mac or *Ctrl* + / on Windows.
- If you need to move the line to another place in the code, press *Shift* + ⌘ + *Down*/*Up* on Mac or *Ctrl* + *Shift* + *Down*/*Up* on Windows. It will also preserve syntactical correctness when moving.

Search navigations

We can use the search navigations to quickly find what we need in our project:

- To find something in path, press *Shift* + ⌘ + *F* on Mac or *Ctrl* + *Shift* + *F* on Windows. To replace something in path, press *Shift* + ⌘ + *R* on Mac or *Ctrl* + *Shift* + *R* on Windows.
- To find all usages of the selected piece of code, press *Alt* + *F7*. To find the usages of the selected piece of code only in the current file, press ⌘ + *F7* on Mac or *Ctrl* + *F7* on Windows.
- To open the last used tool window, press *F12*. To close the currently active tool window, press *Shift* + *Esc*.

Note

To get more information about shortcuts read the online documentation: https://www.jetbrains.com/webstorm/help/keyboard-shortcuts-and-mouse-reference.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.

Summary

In this chapter, you learned about the most distinctive features of WebStorm, which are the core constituents of improving your efficiency in building web applications.

In the next chapter we are going to see how we can build simple web pages with the help of project templates and work with VCS and file watchers.

Chapter 3. Developing Simple Web Pages

Now that you are familiar with some of WebStorm’s smart features, we will focus on creating simple web pages or projects with its assistance.

By the end of this chapter, you will be able to quickly start projects with the help of templates and use your existing codebase inside WebStorm. You will also learn the basics of version control systems, and to use WebStorm for working with transpiled languages like SASS, LESS, or TypeScript.

We are going to study the following topics in this chapter:

- Creating a new project using templates
- Importing an existing project
- Working with a VCS inside WebStorm
- File Watchers

Creating a new project using templates

Whereas in the previous chapter we have created a new project from scratch, we are now going to use some popular templates to help us to start faster. When you use a template, your project is automatically equipped with all the necessary libraries and files. In WebStorm 10, you can create a project using the following templates:

- HTML5 Boilerplate
- Web Starter Kit
- React Starter Kit
- Bootstrap
- Foundation
- AngularJS
- Node.js express app
- PhoneGap/Cordova app
- Meteor app
- Dart

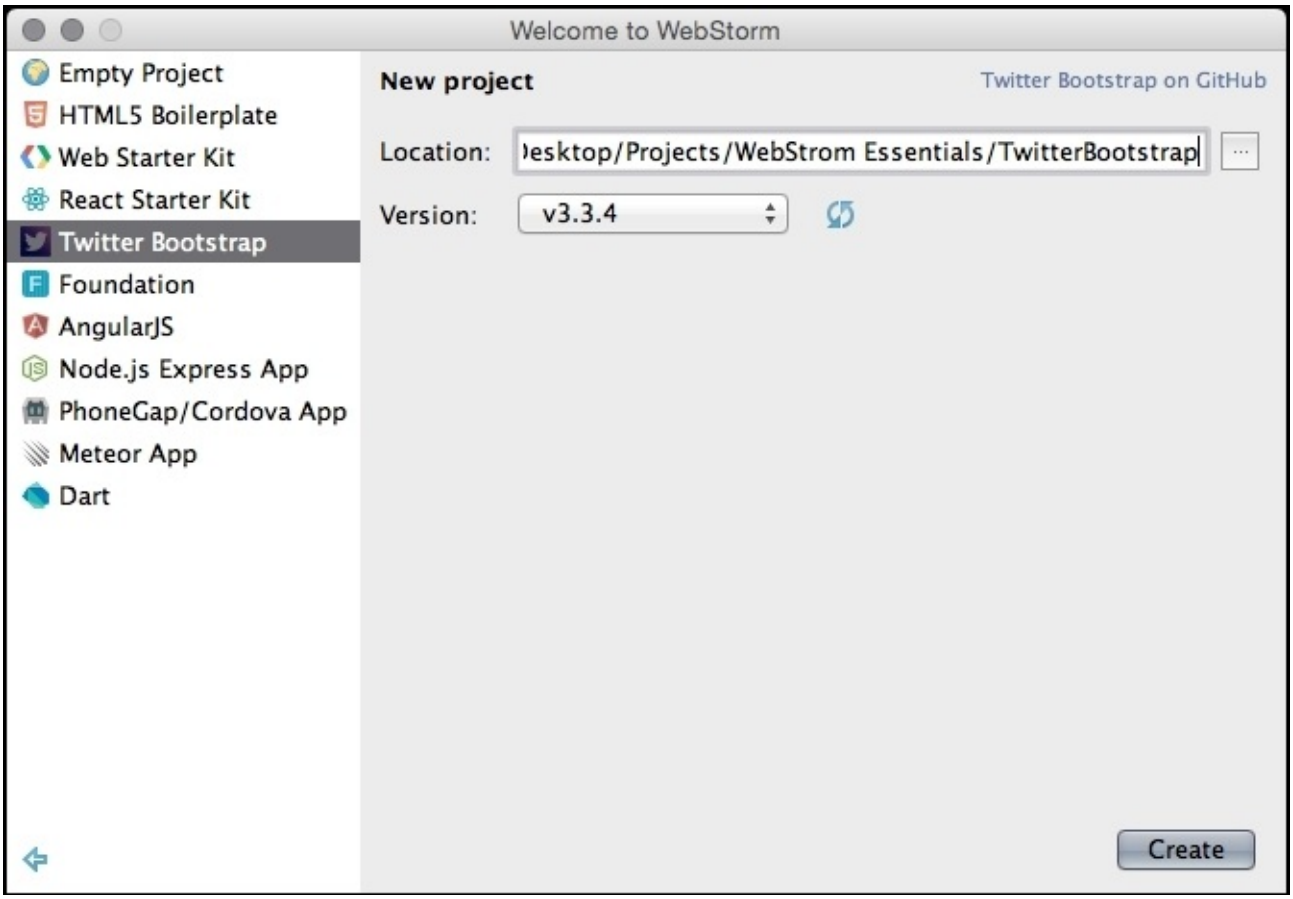
In the following sections, we are going to focus on some of the popular simple frameworks like Bootstrap, Foundation, HTML5 Boilerplate, and the Web Starter Kit. For each of these frameworks, we are going to create or use a simple frameworks example.

Bootstrap

> *"Bootstrap is the most popular HTML, CSS, and JS framework for developing responsive, mobile first projects on the web"*
>
> —http://getbootstrap.com/

Bootstrap is a framework created by Twitter for the faster creation of device-responsive web applications. It can be seen as a collection of CSS classes that can be used to create different elements.

To start a project based on this framework, we have to first choose **New Project** from either the **File** menu or the welcome screen. Then, in the new screen, select **Twitter Bootstrap** as the template. Fill in the desired location for the project, and click **Create**:



This will create a project with the selected version of the framework, and will create a project with its inbuilt files.

Since Bootstrap doesn't come with an example page, we are going to create a simple one by ourselves. In the root project directory, create an `index.html` file and fill in the following code. I strongly advise you to try and enter all the content manually, so that you can get a better understanding of how the autocomplete features help you with your coding:

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

```
<head lang="en">
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-COMPATIBLE" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>WebStorm Bootstrap Template</title>
    <link href="css/bootstrap.min.css" rel="stylesheet">
    <link href="css/main.css" rel="stylesheet">

</head>
<body>
<header class="navbar navbar-default navbar-fixed-top">
    <div class="container">
        <nav role="navigation">
            <div class="container-fluid">
                <!-- Brand and toggle get grouped for better mobile display
-->
                <div class="navbar-header">
                    <button type="button" class="navbar-toggle collapsed"
data-toggle="collapse"
                            data-target="#bs-example-navbar-collapse-1">
                        <span class="sr-only">Toggle navigation</span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                        <span class="icon-bar"></span>
                    </button>
                    <a class="navbar-brand" href="#">WebStorm Essential</a>
                </div>

                <!-- Collect the nav links, forms, and other content for
toggling -->
                <div class="collapse navbar-collapse" id="bs-example-
navbar-collapse-1">
                    <ul class="nav navbar-nav">
                        <li class="active"><a href="#">Home <span
class="sr-only">(current)</span></a></li>
                        <li><a href="#">Authors</a></li>
                        <li class="dropdown">
                            <a href="#" class="dropdown-toggle" data-
toggle="dropdown" role="button"
                               aria-expanded="false">Chapter <span
class="caret"></span></a>
                            <ul class="dropdown-menu" role="menu">
                                <li><a href="#">Chapter 1</a></li>
                                <li><a href="#">Chapter 2</a></li>
                                <li class="divider"></li>
                                <li><a href="#">Chapter 3</a></li>
                            </ul>
                        </li>
                    </ul>

                    <ul class="nav navbar-nav navbar-right">
                        <form class="navbar-form navbar-left"
role="search">
                            <div class="form-group">
                                <input type="text" class="form-control"
placeholder="Search">
```

```
                        </div>
                    </form>
                </ul>
            </div>
            <!-- /.navbar-collapse -->
        </div>
        <!-- /.container-fluid -->
    </nav>
</div>
</header>
<div class="jumbotron presentation">
    <div class="container">
        <div class="row">
            <div class="col-sm-8">
                <h1>WebStorm Essentials</h1>

                <p>
                    Exploit the functional power of WebStorm for faster
building of better JavaScript applications.
                </p>
            </div>
            <div class="col-sm-4">
                <img src="https://placeholdit.imgix.net/~text?
txtsize=47&txt=WebStorm&w=500&h=250"
                     class="img-responsive thumbnail">
            </div>
        </div>
    </div>
</div>
<div class="container highlights">
    <div class="row">
        <div class="highlight-col col-sm-4">
            <div class="highlight-icon glyphicon glyphicon-user"></div>
            <h2>Audience</h2>

            <p>
                This book is intended for web developers with no knowledge
of WebStorm yet experienced in JavaScript, Node.js, HTML, and CSS, and who
are reasonably familiar with frameworks like AngularJS and Meteor.
            </p>
        </div>
        <div class="highlight-col col-sm-4">
            <div class="highlight-icon glyphicon glyphicon-flag"></div>
            <h2>Mission</h2>

            <p>
                The mission of the book is to show the readers that
WebStorm is the number one choice for rapid developing web applications due
to its advanced features and integrationf of a bunch of topical
technologies into itself.
</p>
        </div>
        <div class="highlight-col col-sm-4">
            <div class="highlight-icon glyphicon glyphicon-check"></div>
            <h2>Objectives </h2>
```

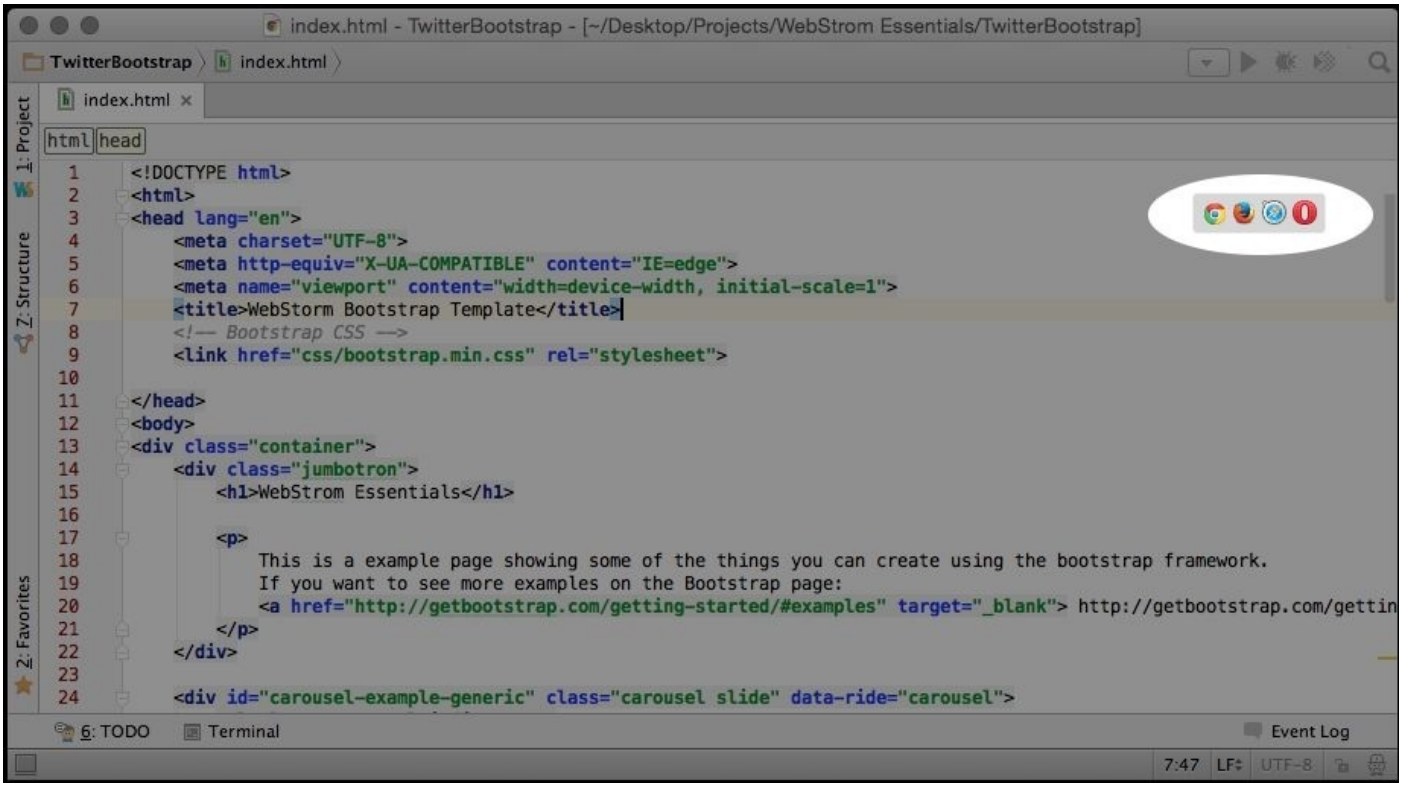
```
            <p>
                To provide info on the newest features of the last version,
and teach how to simplify the web development process using WebStorm
Showcase, and then, to teach the best practices of the cutting-edge
technologies.
            </p>
        </div>
    </div>
</div>
<hr class="feature-divider">
<div class="container">
    <p>
        Lorem ipsum dolor sit amet, consectetur adipiscing elit. Aliquam
neque tellus, gravida vitae elementum sit amet, i porta augue, nec mattis
sapien lacus a nulla. Vivamus in mauris at est porta congue sit amet sed
ipsum. Aenean et dolor dolor. Sed faucibus sem interdum neque euismod, et
tempor lectus eleifend.Pellentesque risus nulla, ornare sit amet semper
vel, congue et justo. Donec a massa commodo, bibendum urna ac,suscipit
ipsum. Sed dapibus quis nisl quis posuere. Vivamus id ipsum suscipit nisl
semper convallis. Fusce lobortis, metus et interdum mattis, eros eros
tincidunt ipsum, et faucibus elit eros vel ante. Nullam vitae mattis
ligula, id scelerisque lacus. Donec mattis odio quis sem facilisis dapibus.
    </p>

    <p>
        Sed nec lobortis dui. Vestibulum a lectus eleifend, viverra metus
eu, ornare sem. Curabitur eu tellus massa. Sed vestibulum dolor sed sodales
ullamcorper. Nullam auctor maximus scelerisque. Praesent vulputate diam vel
scelerisque commodo. Nulla maximus lectus malesuada vehicula aliquet.Nunc
eleifend purus sit amet ante rhoncus maximus. Maecenas orci nisi, pretium
in maximus quis, finibus sed sem.
    </p>

    <p>
        Integer lorem odio, feugiat eu faucibus eget, varius vulputate
tellus. Duis convallis nibh ac ligula congue venenatis. Aenean dapibus
condimentum quam, sed iaculis leo feugiat sit amet. Nunc eu nibh vestibulum
tellus tincidunt pharetra at sed massa. Vestibulum tempor aliquet neque, et
auctor turpis sollicitudin et. Aliquam acvarius ipsum, sit amet vehicula
tortor.Nullam ac est nec ante scelerisque tincidunt.
    </p>
</div>
<footer class="footer copy">
    <div class="container">
        <p class="text-muted pull-right">Copyright &copy; 2015. All rights
reserved</p>
    </div>
</footer>
<script
src="https://ajax.googleapis.com/ajax/libs/jquery/1.11.2/jquery.min.js">
</script>
<script src="js/bootstrap.min.js"></script>
</body>
</html>
```

After that, we need to create the `main.css` file, and add the following rules to it:

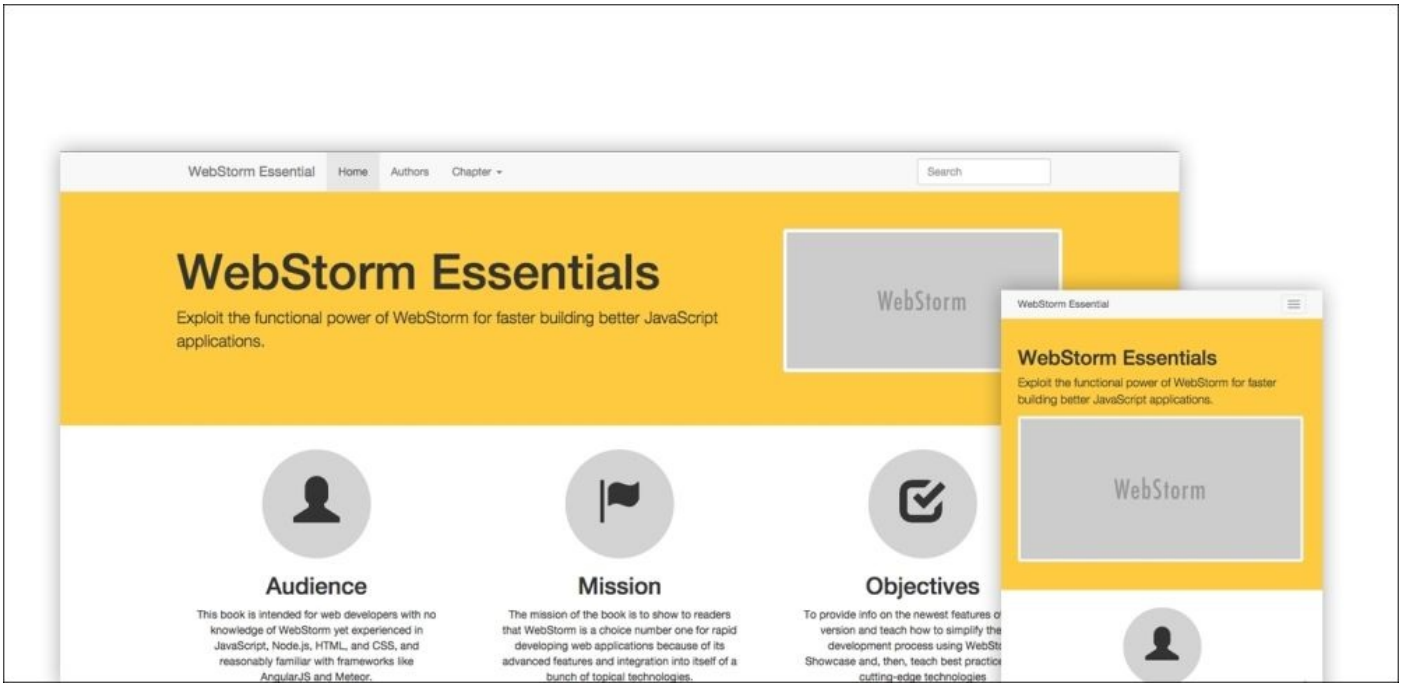
```
.presentation{
    margin-top: 30px;
    background: #feca5c;
}

.highlight-col{
    text-align: center;
    padding: 0 40px;
}
.highlight-icon{background: lightgray;
    width: 140px;
    height: 140px;
    font-size: 60px;
    border-radius: 50%;
    padding-top: 35px;
}
.feature-divider{
    margin: 50px 0;
}

.copy{
    background: #f8f8f8;
    padding: 20px 0;
}
```

You can see that our CSS code is minimal, because by using Bootstrap, we get all the necessary styles. The only thing that we have to use is the HTML markup that Bootstrap recommends.

Now that we have created the page, we can click the browser icons to open the file inside the selected browser. These browser icons appear when you hover your mouse on the top-right side of the `index.html` file, shown as follows:



WebStorm will create a local server, and open the page in the selected browser. We will now see the following page in the browser:



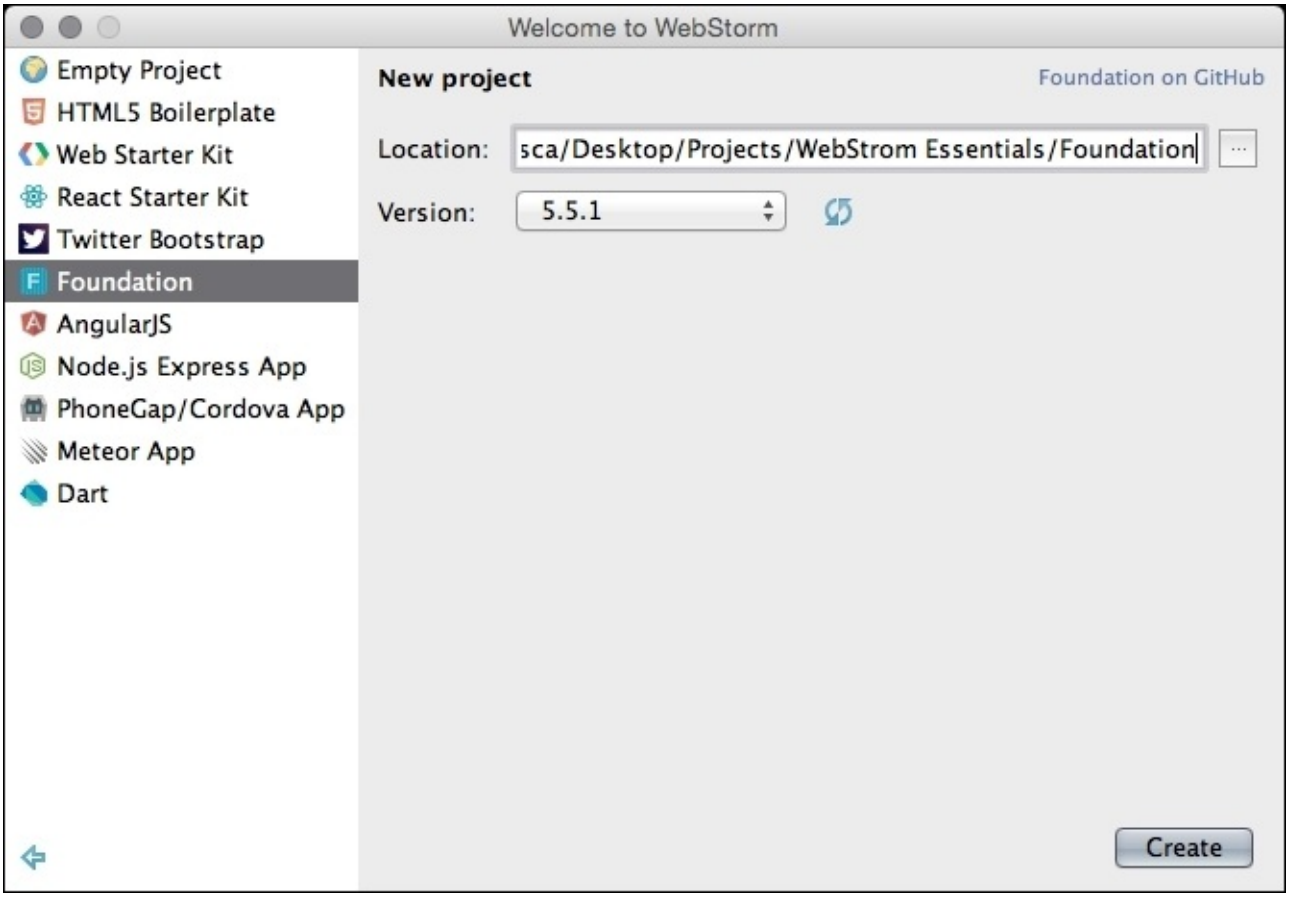
We have created a simple page using some of the styles that come with the Bootstrap framework. One of the big advantages that come with this framework is that our page is responsive and suited for a whole range of devices.

Foundation

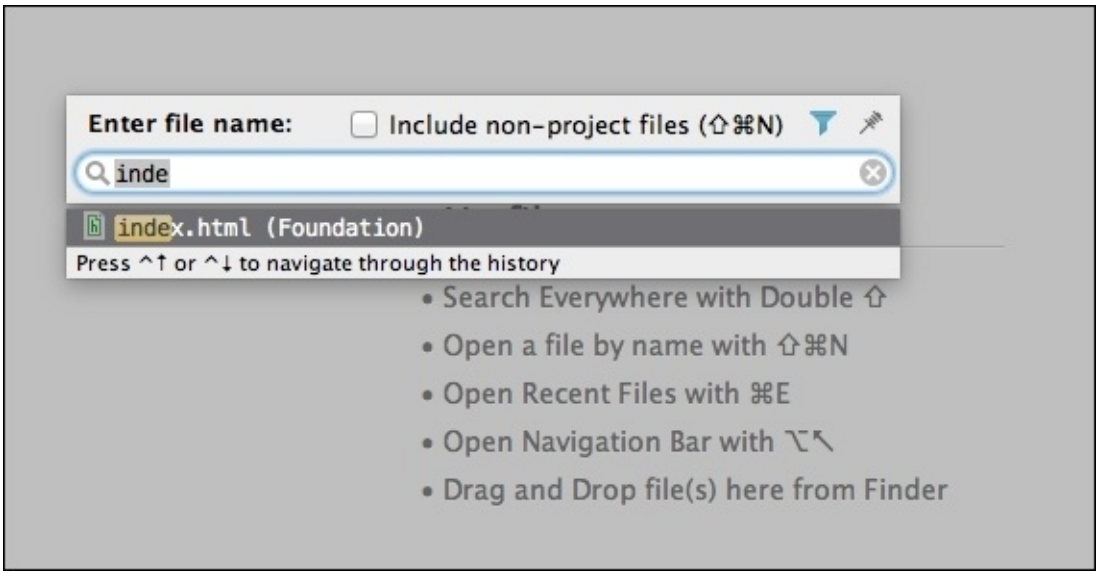
> *"The most advanced responsive front-end framework in the world."*
>
> —http://foundation.zurb.com/

Foundation is a framework similar to Bootstrap that provides a responsive grid and a set of UI elements that we can use to jumpstart our project creation.

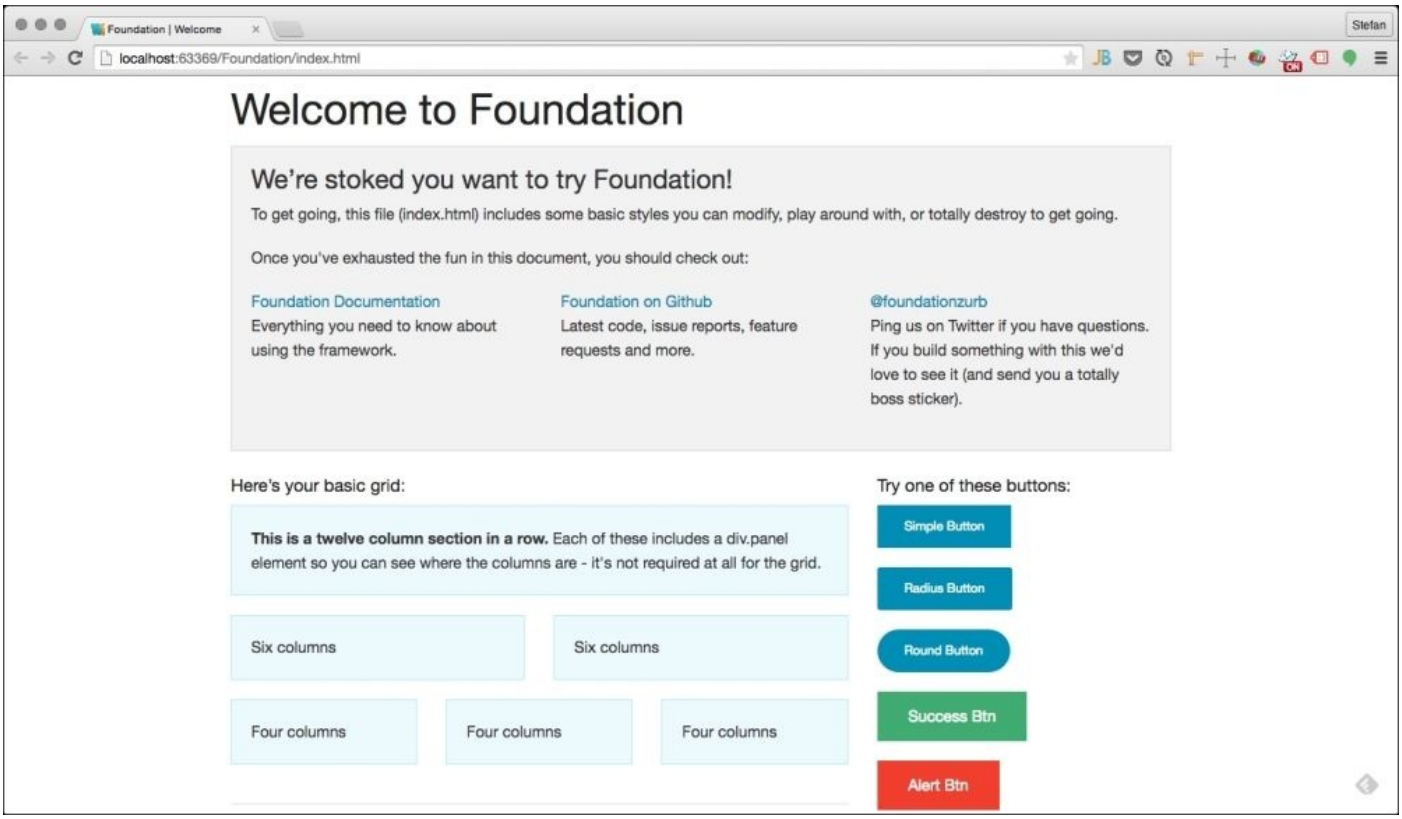
To start a project, we go through the same steps as for the Bootstrap project; but this time, we select **Foundation** as the project template, as seen in the following screenshot:



Foundation comes with a previously-created example, so if we want to see an example, we have to open the `index.html` file inside WebStorm. A quick way to open files inside a project is by using the open a file by name dialog: press *Shift* + ⌘ N on Mac or *Ctrl* + *Shift* + *N* on Windows, and start typing the name of the file that you want to open.



In the opened `index.html` file, we have to click the browser icon, just like we did in the Bootstrap example, to see the Foundation example page in the browser.

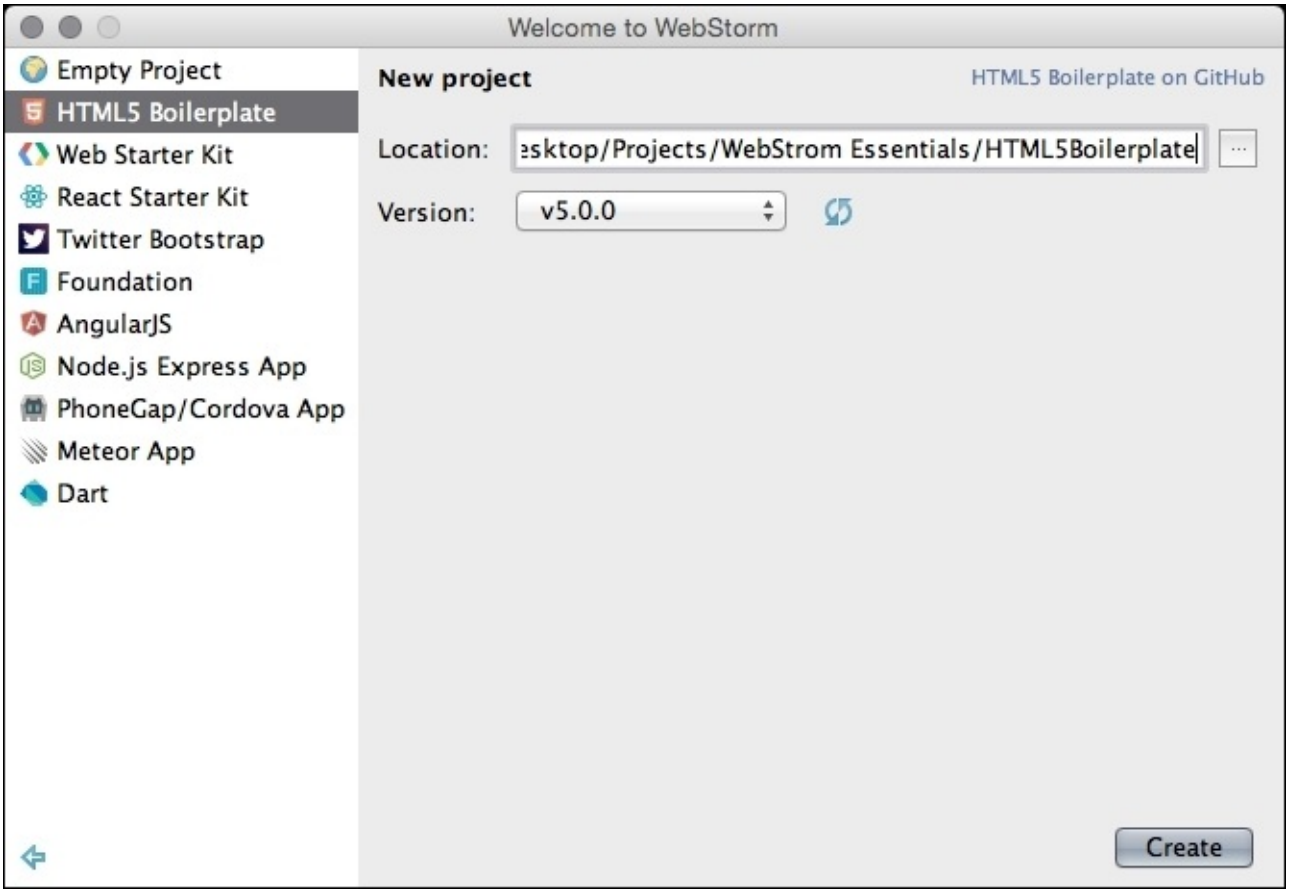


HTML5 Boilerplate

> *"The web's most popular front-end template*
>
> *HTML5 Boilerplate helps you build fast, robust, and adaptable web apps or sites. Kick-start your project with the combined knowledge and effort of 100s of developers, all in one little package."*
>
> —https://html5boilerplate.com

HTML5 Boilerplate is a template that helps you to quickly start your project by creating some of the most important files and importing some libraries like `normalize.css`, `jQuery`, and `Modernizer`.

To create a new project based on this template, we have to choose **New Project** from either the **File** menu or the welcome screen. Then, in the new screen, select **HTML5 Boilerplate** as the template. Fill in the desired location for the project and click **Create**:



HTML5 Boilerplate comes with an example page; so, to see it, we have to open the `index.html` file inside WebStorm. A quick way to open files inside a project is by using the open a file by name dialog, press *Shift* + ⌘ + *N* on Mac or *Ctrl* + *Shift* + *N* on Windows, and start typing the name of the file that you want to open.

Once we have opened the `index.html` file, we can click the browser icons to open the file inside the selected browser. These browser icons appear when you hover the mouse on the top-right side of the opened file.

You have now created a new project based on the popular HTML5 framework, and you

can start changing and adding new files to expand your project.

Web Starter Kit

The Web Starter Kit is a boilerplate and tooling template for multi-device development. It is presented on the developers' website as follows:

> *"Your starting point for building great multi-device web experiences Start your project with the Web Starter Kit and ensure you're following the Web Fundamentals guidelines out of the box."*
>
> —https://developers.google.com/web/starter-kit/

Now we are going to start a new project again, but this time, it will be one based on the Google framework. To do that we are going to select **New Project** from the **File** menu or the welcome screen, and select **Web Starter Kit** as the template.

This template has more features and dependencies, so we need to install the required packages and Gulp. We are going to install Gulp globally so that it will be available to all the projects. To do this, we simply open the WebStorm internal terminal and run the following:

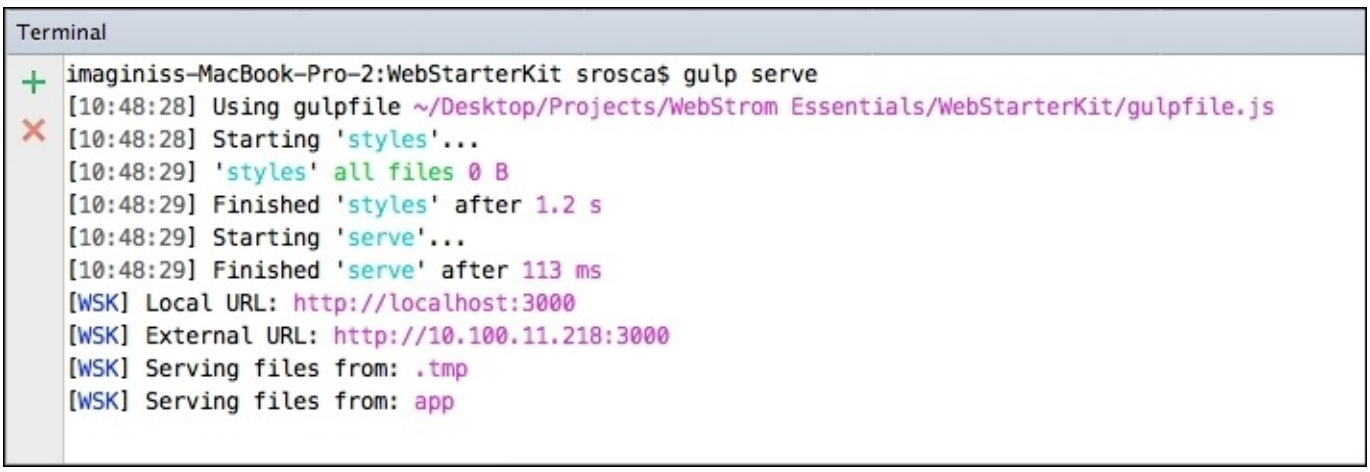
```
npm install
npm install gulp -g
```

Once you have all the packages installed, you can start developing. The Web Starter Kit has three development phases that you are going to use.

The first one is the local server development phase. To get in this phase, you have to run the following command in the terminal:

```
gulp serve
```

The following screenshot displays the output of the preceding command:



Once you run this command, you should see the preceding message in the terminal and your browser opens with the page. Make a note of the `External URL` as you can use this to test your page across multiple devices. Just point your phone or tablet to this URL, and the browsing experience will be synced as you work.

In this phase, the following tools are at work:

- **Live Reload**: This watches the files for changes and automatically loads them in the browser
- **Browser Sync**: This synchronizes the opened page across multiple browsers and devices
- **JSHint**: This scans your JavaScript code and checks for possible problems
- **Sass compile**: Any changes you make to the Sass files will be compiled into CSS after your page is reloaded with Live Reload
- **Automate prefixing**: This automatically appends any vendor prefixes that are necessary in your styles

The second phase is the production build phase. In this phase, the production version of your page is built. To run this phase, execute the following command in the terminal:

```
gulp
```

The following steps are performed in the build phase:

1. **Build Styles**: This compiles the Sass and runs the autoprefixer.
2. **JSHint**: This scans your JavaScript code and checks for possible problems.
3. **HTML build**: This minifies the HTML and the JavaScript.
4. **Automate prefixing**: This automatically appends any vendor prefixes that are necessary in your styles.

The third phase is the production build test. In this phase, we test our production build to make sure that everything works as expected. A production version of your site is built in this phase, and is then opened in a browser. To run this phase, you have to type the following command in the terminal:

```
gulp serve:dist
```

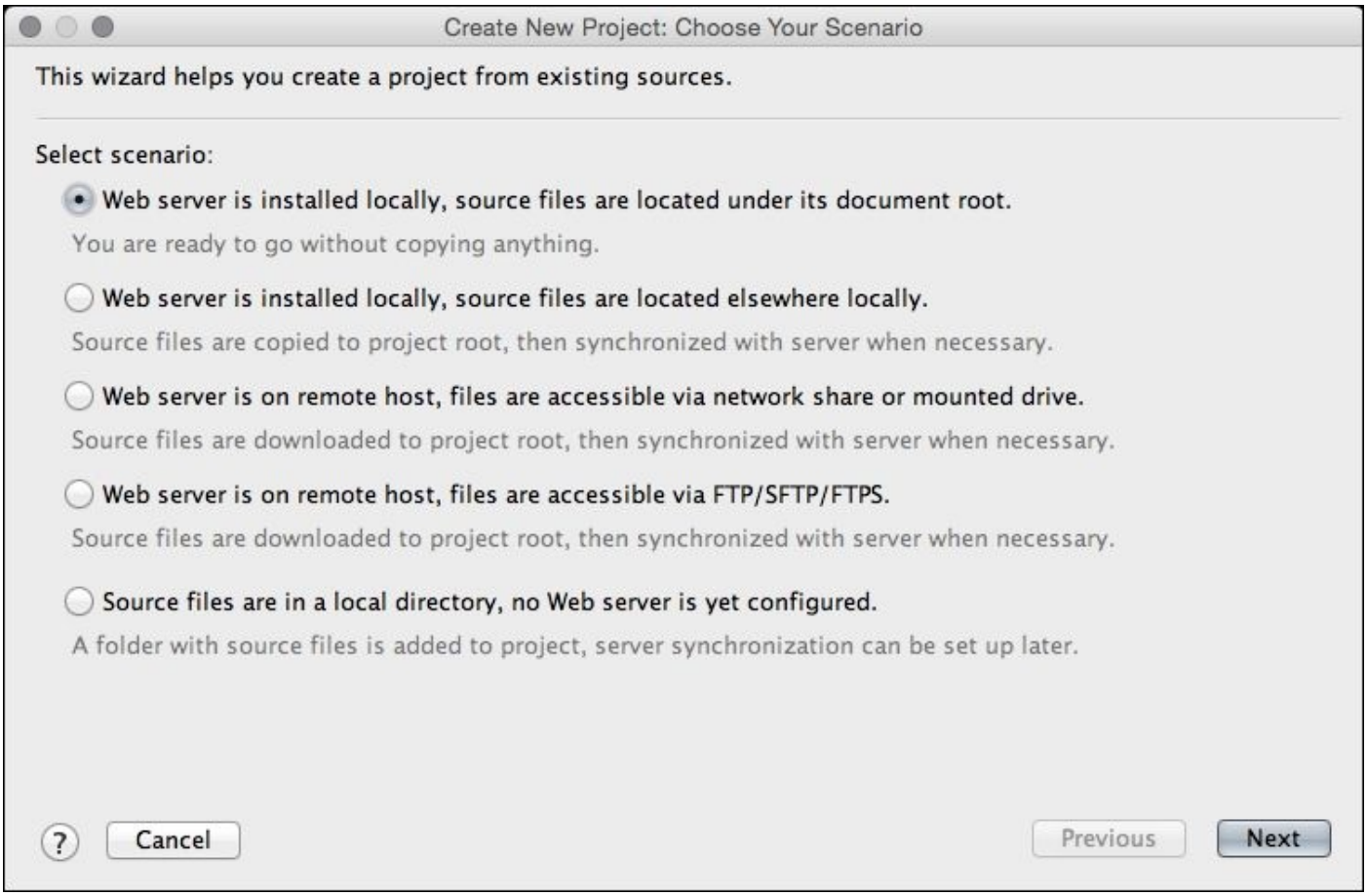
As you can see from the example pages, the Web Starter Kit is a really powerful platform that helps you quickly start building a web application, especially for cross-platform applications and responsive websites.

Importing an existing project

In the previous sections, we have created new projects based on some of the popular templates available in WebStorm. In this section, we are going to focus on working with the existing projects, and we are going to explore two ways of creating a project using the existing files or a versioning control system.

Importing from existing files

Sometimes, you have existing projects on your drive that you want to edit in WebStorm. To do that, we simply have to select **Create Project From Existing files** from either the welcome Screen or the **File** menu. Once we select this option, we are presented with the following screen where we have to select a scenario:

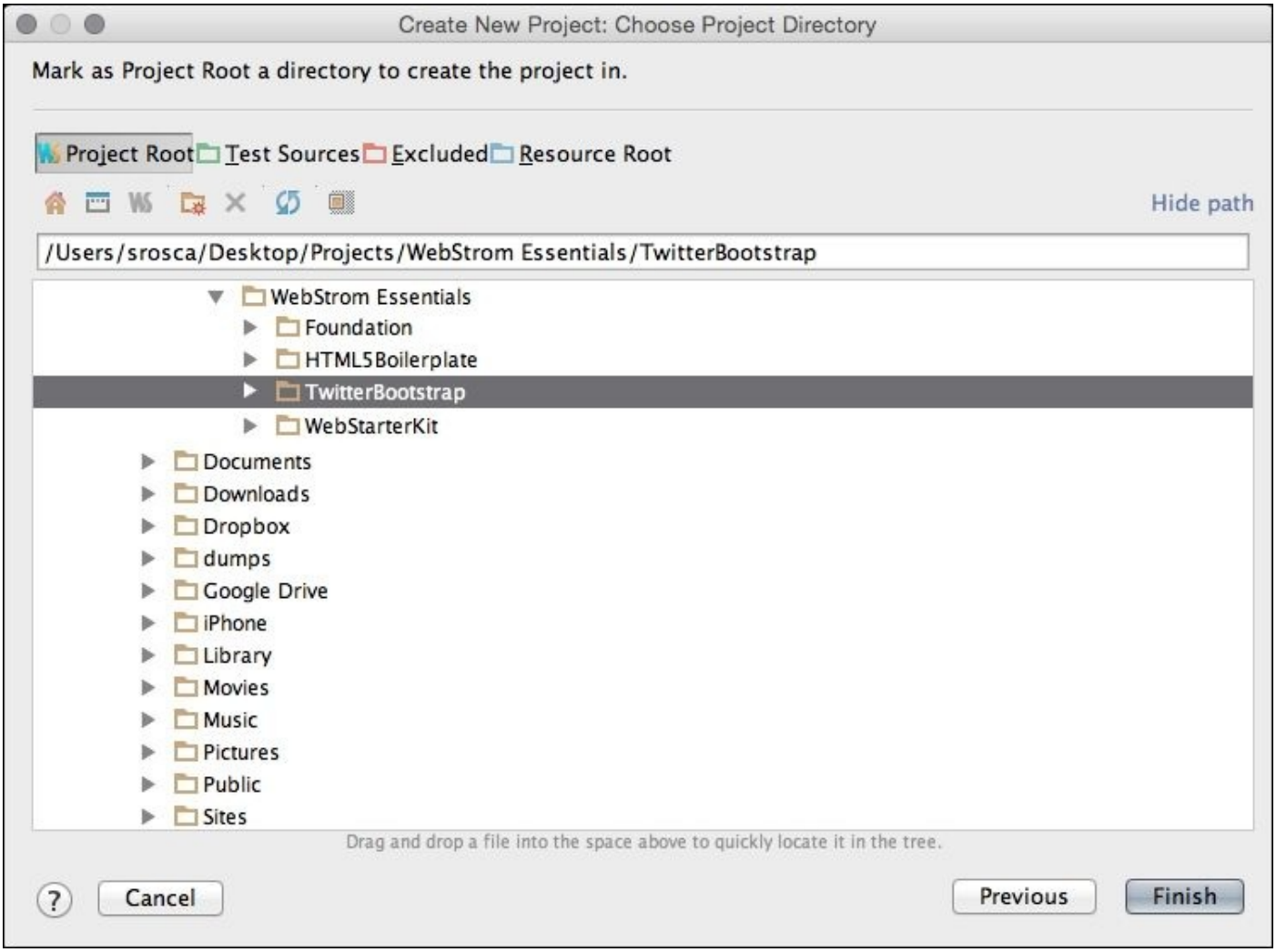


In the preceding dialog, we are presented with the following scenarios:

- **Web Server is installed locally, source files are located under its document root**: In this scenario, we presume that we have installed a local server and our project is inside the server document root (for example, document root in Apache Mac or httpdocs on an XAMPP installation).
- **Web Server is installed locally, source files are located elsewhere locally**: In this scenario, we have installed a local server and our project is outside the server document root. The files will be copied to the server and synchronized when necessary.
- **Web server is on remote host, files are accessible via network share or mounted drive**: We select this option to make WebStorm copy files to a local drive via the network from a remote server, and then set up a project around them.
- **Web server is on remote host, files are accessible via FTP/SFTP/FTPS**: Select this option to make WebStorm download files from a remote server via the FTP, SFTP, or FTPS protocol (for example a hosting account).

- **Source files are in a local directory, no Web server is yet configured**: Select this option if you want to work with files in a certain local directory, without using any Web server.

We are going to use the last version mentioned in the previous list since we don't want to use a specific server at the moment. On selecting the version, we will be presented with the screen where we have to select the project directory page, as shown in the following screenshot:



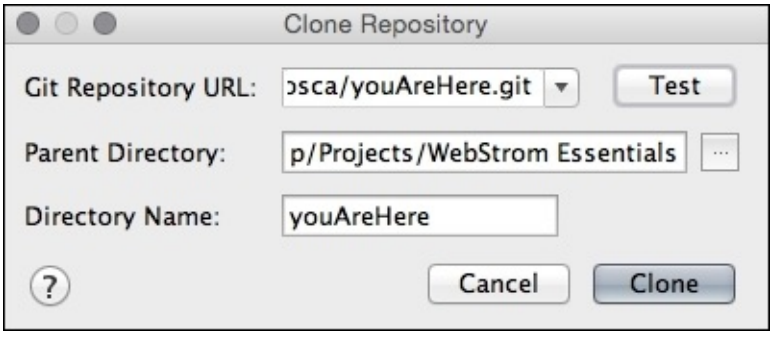
Select the folder that contains your project (in this case, we are going to use the previously-created `TwitterBootstrap` folder), mark it as **Project Root,** and then click **Finish**.

Importing an existing project from VCS

In this section, we are going to explore some of the ways for importing an existing project from a VCS. We are going to import an existing Git project.

To do so, we have to select **Check out from version control** from either the welcome screen or the VCS. In this demo, we are going to check out a project from **Git,** so select that option, as shown in the next screenshot:

Now specify the **Git Repository URL**. We have created a GitHub project for this purpose; so go ahead and fill this with https://github.com/srosca/youAreHere.git, and click on **Clone**. You can also **Test** your connection to the Git repository in this screen.



This will clone the Git repository inside the specified directory, and then prompt you to

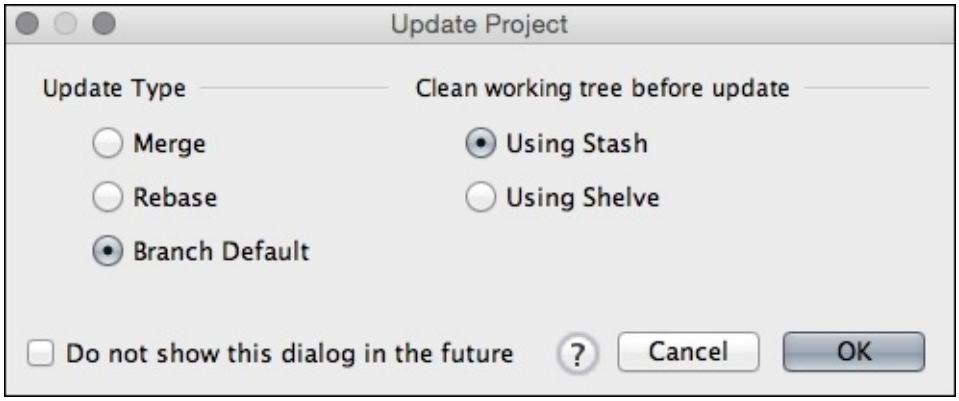
open the project.

In this example, we have focused on creating a project from a Git repository, but the same steps can be used with other versioning systems like GitHub, CVS, Git, Mercurial, and Subversion.

Working with VCS inside WebStorm

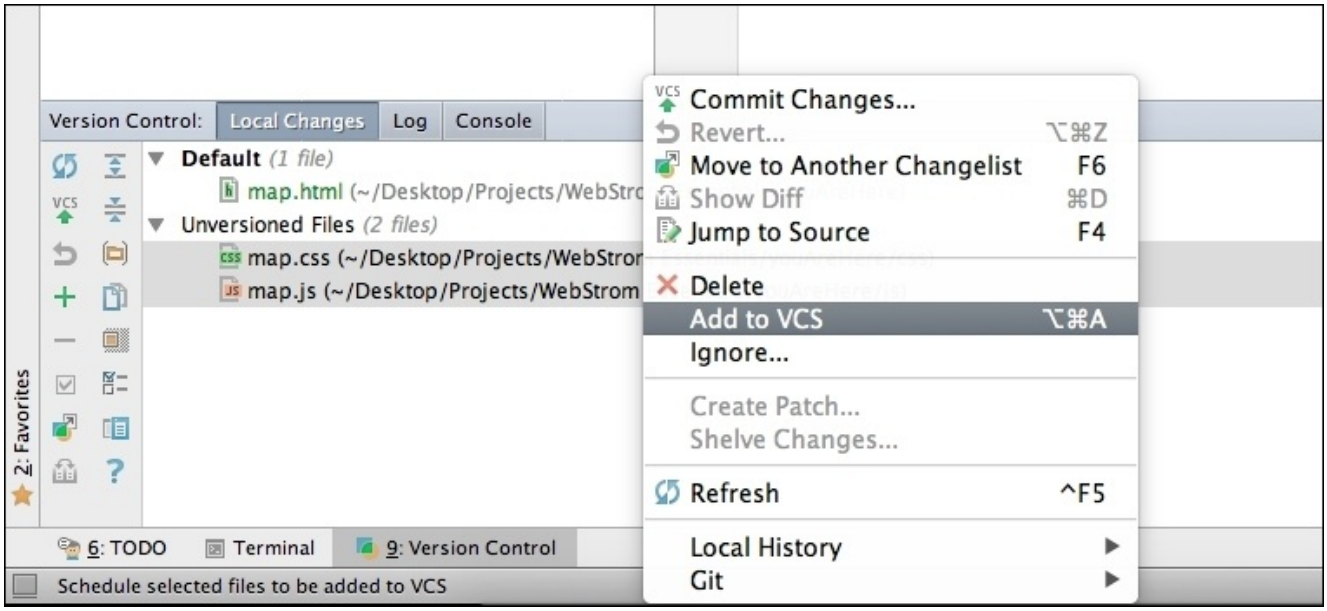
If you have a project linked to a VCS, you can perform all the necessary actions inside WebStorm. We will now go through some of the most-used tasks:

- **Update**: We can update the project by selecting **Update Project** from the **VCS** menu, as seen in the following image:

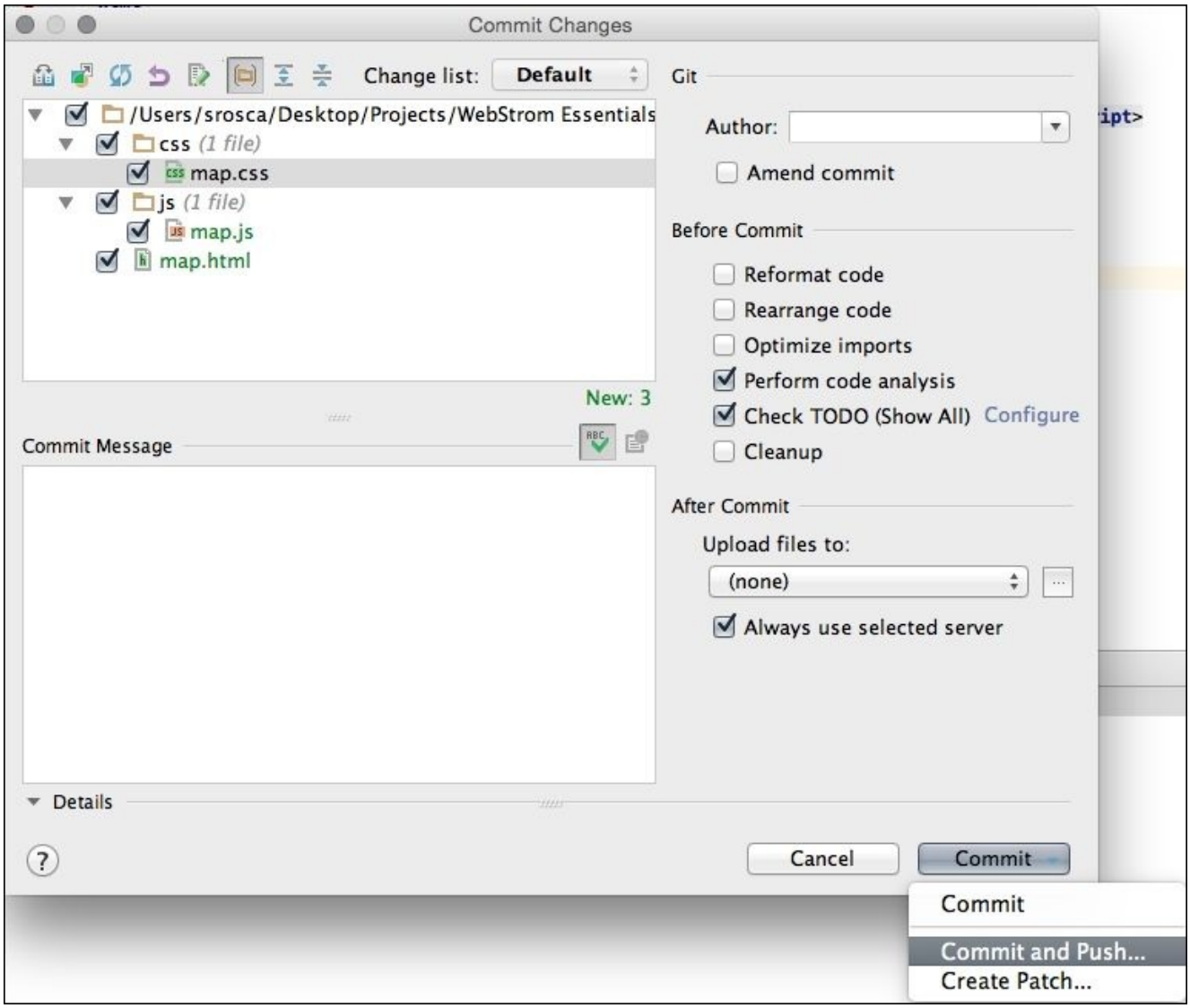


  We will be presented with a dialog to select the update strategy. Select **Branch Default** and **Using Stash**; click on **OK**. In this way, WebStorm will use the default command for the applied branch, and use and save the changes in a Git stash.

- **Add**: Before you commit files, we must add them to the current change list. We can add files by going to the **Local Changes** tab in the **Version Control** section. Under that, select the files, and then select **Add to VCS** from the context menu:



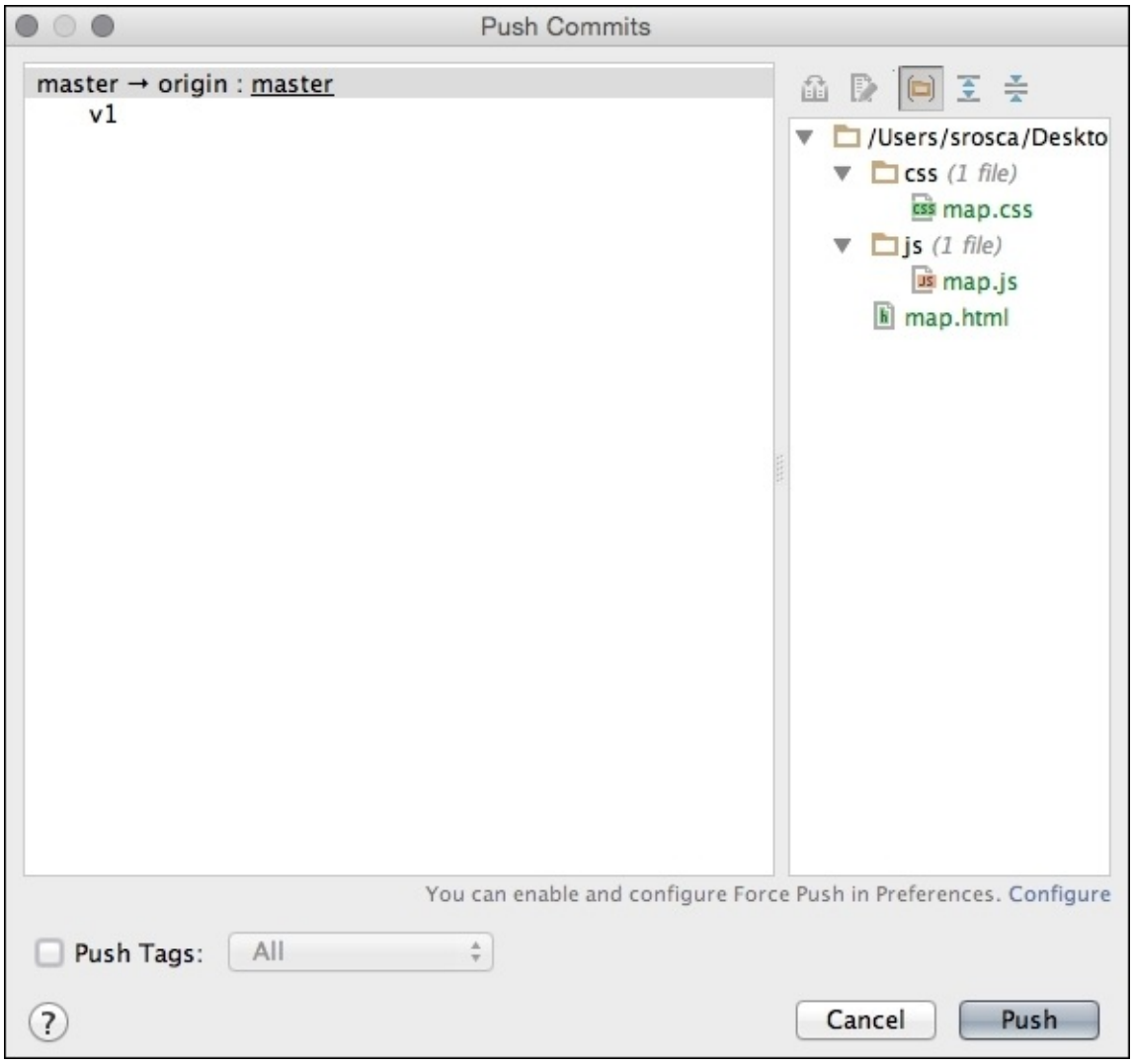
- **Commit**: To commit the files, we have to select **Commit Changes** from the **VCS** menu or use the shortcuts ⌘ + *K* or *Ctrl* + *K*:



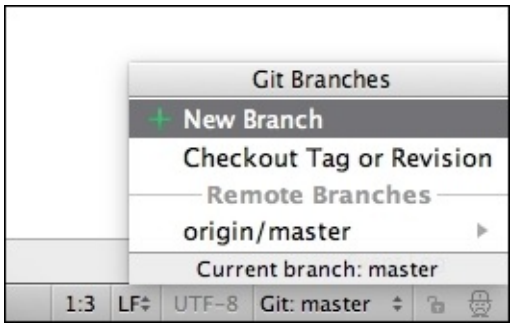
If we have any files selected in the **Version Control** section, then the commit will have only these files preselected.

We can also select **Commit and Push…** to commit and then push the files.

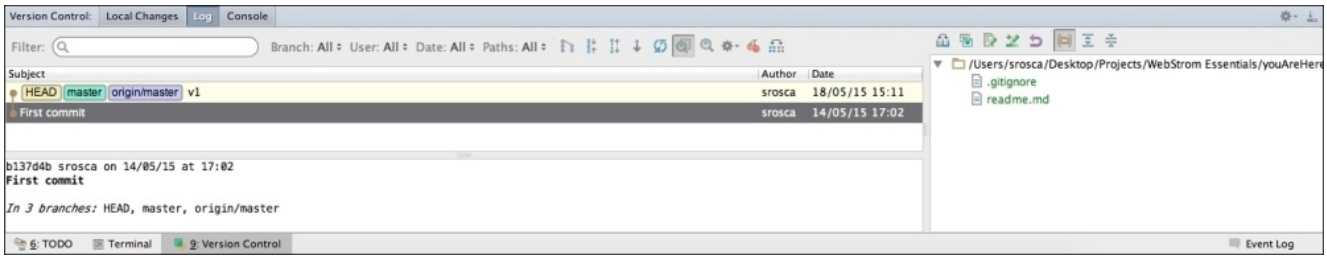
- **Push**: If we have committed changes, we can do a push by going to **VCS** | **Git** | **Push**:



- **Branches**: WebStorm displays the active branch at the bottom-right corner of the screen. We can also perform all the branch-related tasks (for example, creating a new branch) in this menu:



- **History**: In WebStorm, we always have access to a powerful history viewer were we can see all the commits and branches. We can access this in the **Version Control** section, under the **Log** tab:



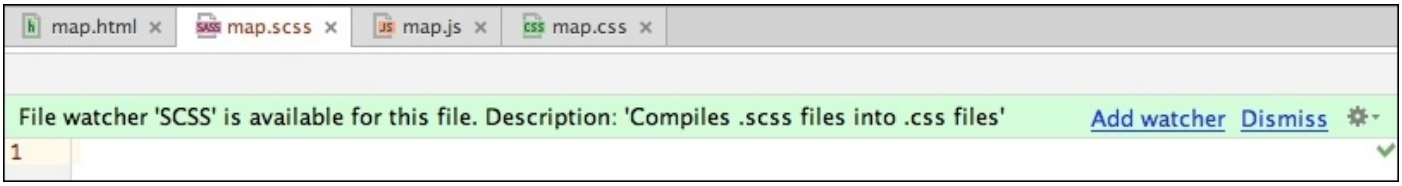
One thing to remember about WebStorm and Git is that all the tasks performed through the UI are, in fact, `git` commands that WebStorm runs in the background. You can see these commands and all the actions performed in the **Console** tab of the **Version Control** section.

We have gone through the Git integration system since it is among the most used ones, but WebStorm works in a similar way with other VCS systems as well.

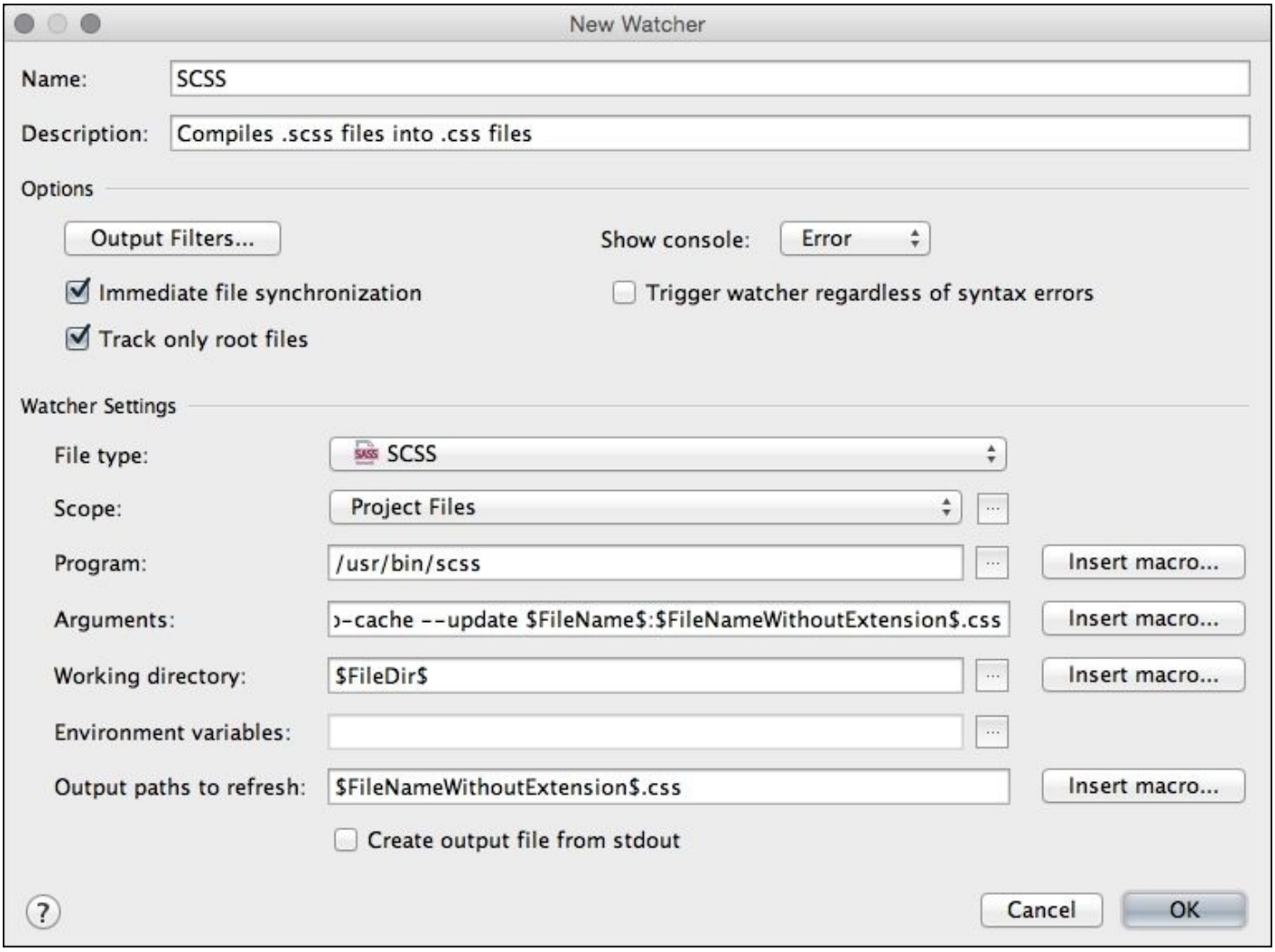
File Watchers

In the current development cycle, transpiled languages have an ever increasing impact. WebStorm can help you work easily with these files with the help of watchers.

We are going to start by creating a `map.scss` file in the `CSS` folder. Once this file has been created and opened, WebStorm will inform us that a watcher is available, and ask us to add a watcher. Copy the code from the `map.css` file inside `map.scss`, and then click on **Add watcher**:



Once we click on the **Add watcher** button, we will be taken to the **New Watcher** dialog where we choose the settings for this process:



Once we start the watcher, WebStorm will watch for changes in the `scss` file and translate

it into CSS code.

WebStorm supports integration with various third-party transpilers that perform the following:

- Translate Less, Sass, and SCSS source code into CSS code
- Translate TypeScript and CoffeeScript source code into JavaScript code, possibly also creating source maps to enable debugging
- Compress JavaScript and CSS code

Note

Note that WebStorm does not contain built-in transpilers, but only supports integration with the tools that you have to download and install outside WebStorm.

You can see a list of the watchers associated with the current project in the **Preferences** | **Tools** | **File Watchers** dialog.

Summary

In this chapter, you have learned how to use WebStorm to quickly start new projects with the use of templates, and also how to use the existing code bases in the editor.

In the next chapter, we are going to learn to use package managers and build systems by means of WebStorm’s built-in features.

Chapter 4. Using Package Managers and Build Automation Tools

In the previous chapter, we focused on building simple webpages with the help of the WebStorm templates and on using the existing codebases inside the editor.

In this chapter, you are going to learn how WebStorm can help us when working with some of the package managers and build tools available for the development workflow. You will also learn how to set up and use the following:

- Node.js
- NPM
- Bower
- Grunt
- Gulp

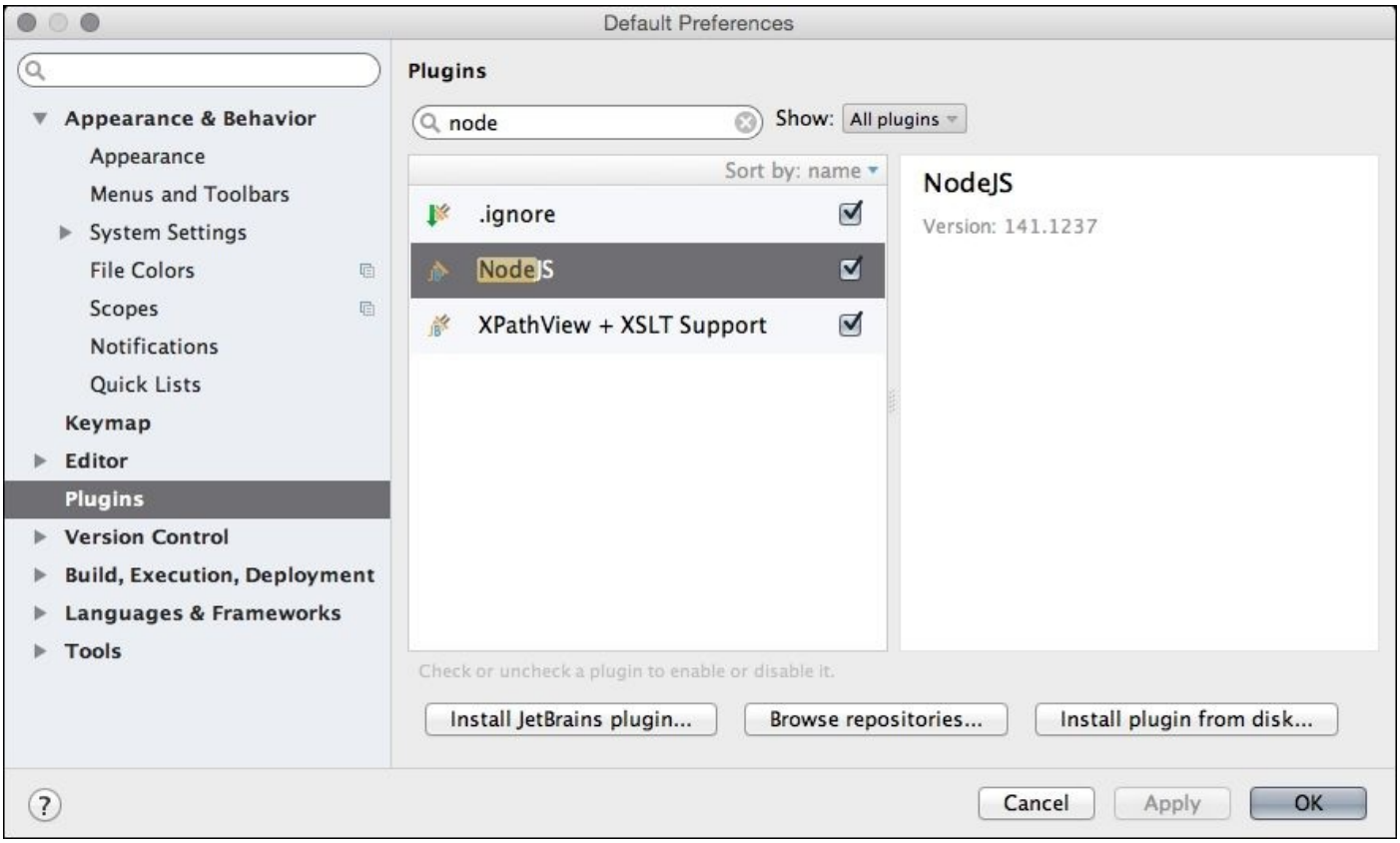
At the end of this chapter, you will be able to download, install, configure, and use package manager tools like NPM and Bower, and task runners like Grunt and Gulp.

Node.js

All the tools that are required in this chapter run on Node.js, so in the first part, we are going to focus on making sure that everything is set up correctly.

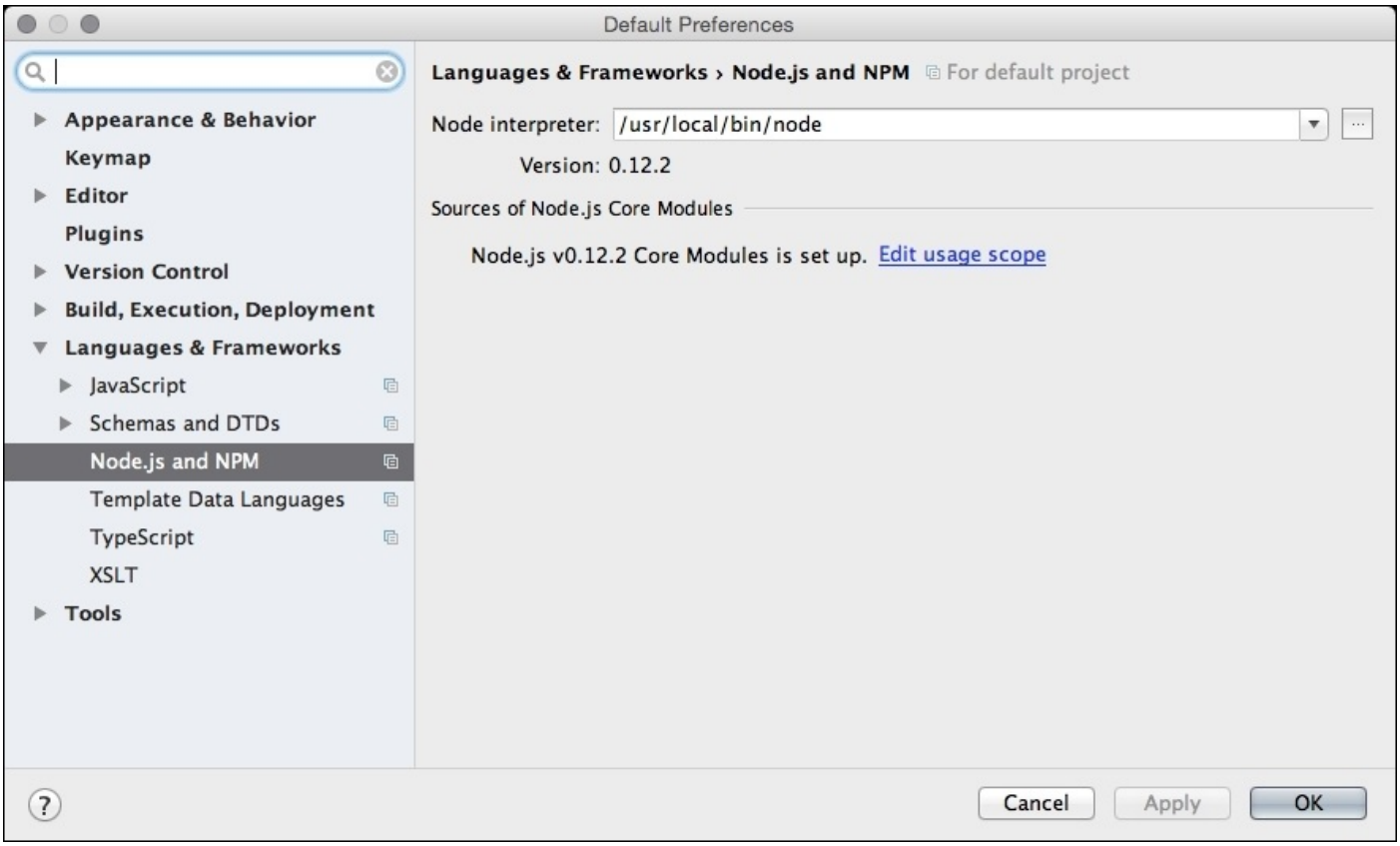
First, we need to check if Node.js is installed. Running the node `node -v` in the terminal window should return the version of node that we have installed. If you don't get the version number, it means that we need to download and install Node.js. Follow the instructions given at https://nodejs.org/#download.

Once we are sure that the node is up and running, we need to check that WebStorm is set up to work with it. Go to the **Preferences** | **Plugins** dialog, and make sure that the **NodeJS** plugin is activated:



When the plugin is activated, WebStorm will create a settings page for the node and add run/debug configurations.

The settings page, which can be accessed at **Preferences** | **Languages & Frameworks** | **Node.js and NPM,** allows us to set the node interpreter and the sources for the core modules, as seen in the following screenshot:



If you don't have the sources configured, you need to click on **Configure** and download the sources in the next dialog. This enables code completion, reference, and debugging for the node core modules like `fs`, `path`, `http`, and others.

Once we have the node installed on our system, we can move to the next section where we are going to learn how to use NPM inside WebStorm.

Using the Node Package Manager to install node packages

NPM is a package manager that comes with the node, and enables us to quickly install node packages, either globally or for the current project.

From Node.js version 0.6.3 onwards, NPM comes bundled and installed automatically with the node, so we don’t have to install anything. We can find all the available packages at https://www.npmjs.com/; each package has a description page with all the necessary information.

Installing a package globally

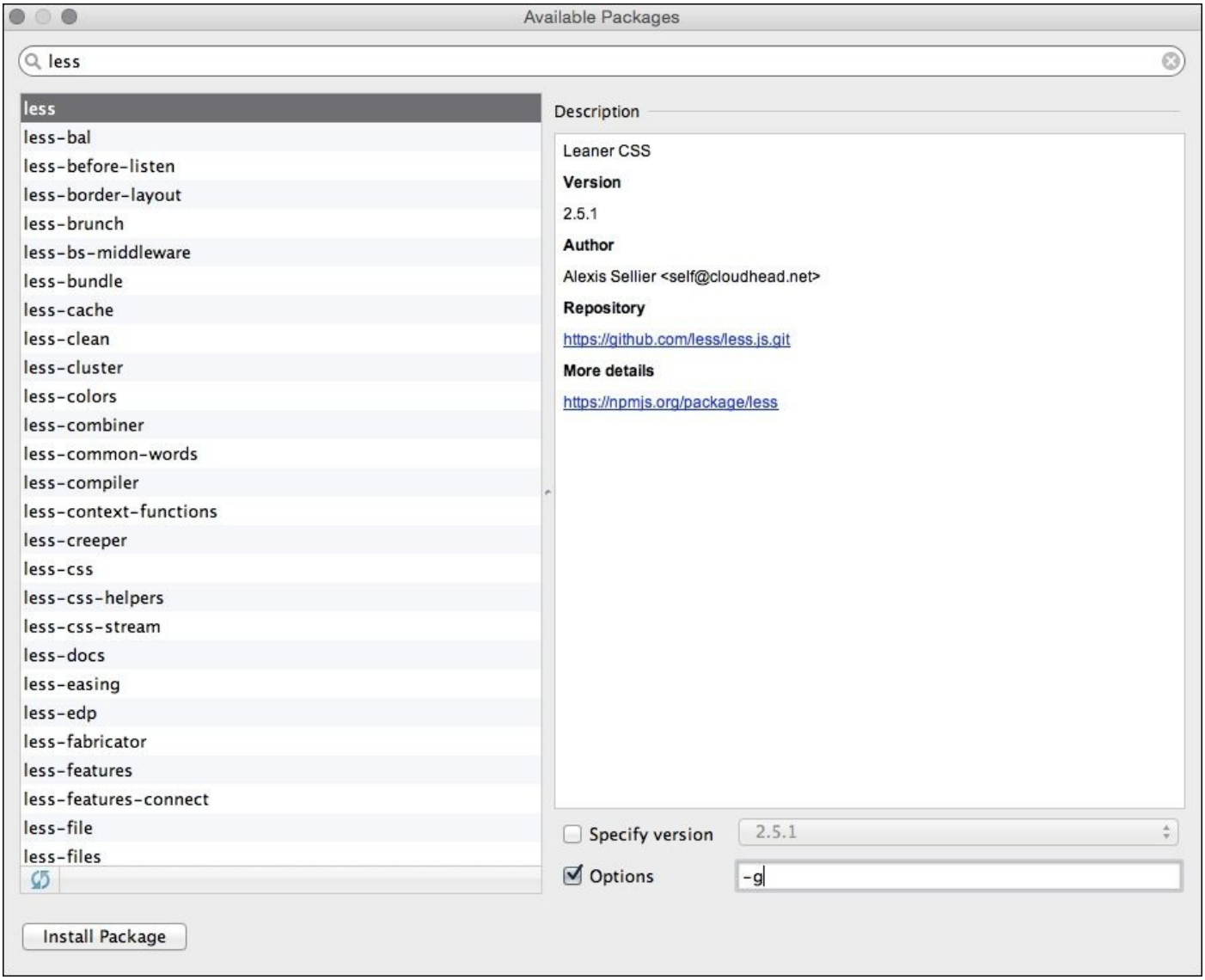
Global installation makes a package available at the system level so that it can be used in all projects. Usually, we install the tools that we need to use on multiple projects like transpilers, compressors, or test runners globally, and these are used from the command line.

To install the package globally, we have two options:

- Run the installation from the command line with the global option:

  ```
  npm install -g <package>
  ```

- On the **Node.js and NPM** settings page, select the install icon + from the **Packages** section. In the new screen, select the package you want to install, and type `-g` in the option field:



Installing a package in the project

A package installed in the project is available only for the current project. When installing a package locally, we have to create a `node_modules` folder manually, or initialize `npm` in the current project to avoid unexpected results. That is because NPM searches up the folder tree for an initialized project, and installs the package there if it can find one.

To initialize the project with the NPM, we have to run `npm init` in the terminal window. This will ask you some questions about your project like the name, version, description, author, and so on, and then create the `package.json`.

To install the package in the project, we have two options:

- Run the installation from the command line:

  ```
  npm install <package>
  ```

- On the Node.js and NPM settings page, select the install icon + from the packages section. In the new screen, select the package you want to install.

The only difference from the globally installed packages is that we don't use the `-g` option anymore.

Note

When we have an initialized project, we can also specify and save the installed packages as a dependency or as a development dependency in the `package.json` file. To do that, we have to use the `--save or --save-dev` option in either the terminal command or the option field in the settings dialog.

Installing project dependencies

If you have a project or have downloaded an existing one, you have to run `npm install` from the command line to install the dependencies. This will install all the modules specified in the `package.json` file.

Using Bower

As mentioned earlier, NPM has a package manager for the node modules and tools. Similarly, we need a package manager for the web as well. This manager is known as Bower, a tool that allows us to install all the libraries, frameworks, and packages that we need for our web project.

The first thing that we need to do before using Bower is to install it. We are going to use NPM and install it as a global package so that it is available for all our projects. So, open the terminal and type the following command:

```
npm install -g bower
```

Now that we have Bower installed, we can use it to search and install the packages that we need. To search for packages, we can use the online tool available at http://bower.io/search/, and search from the terminal by using the `bower search <keyword>` command.

Before we use Bower to install packages, we have to initialize the project so that we can save our dependencies.

Run `bower init` in the terminal, and go through the questions to generate the `bower.json` file.

Once we have initialized Bower to install the packages, we have two options:

- The first option is to run the installation command in the terminal, as follows:

  ```
  bower install <packageName or source>
  ```

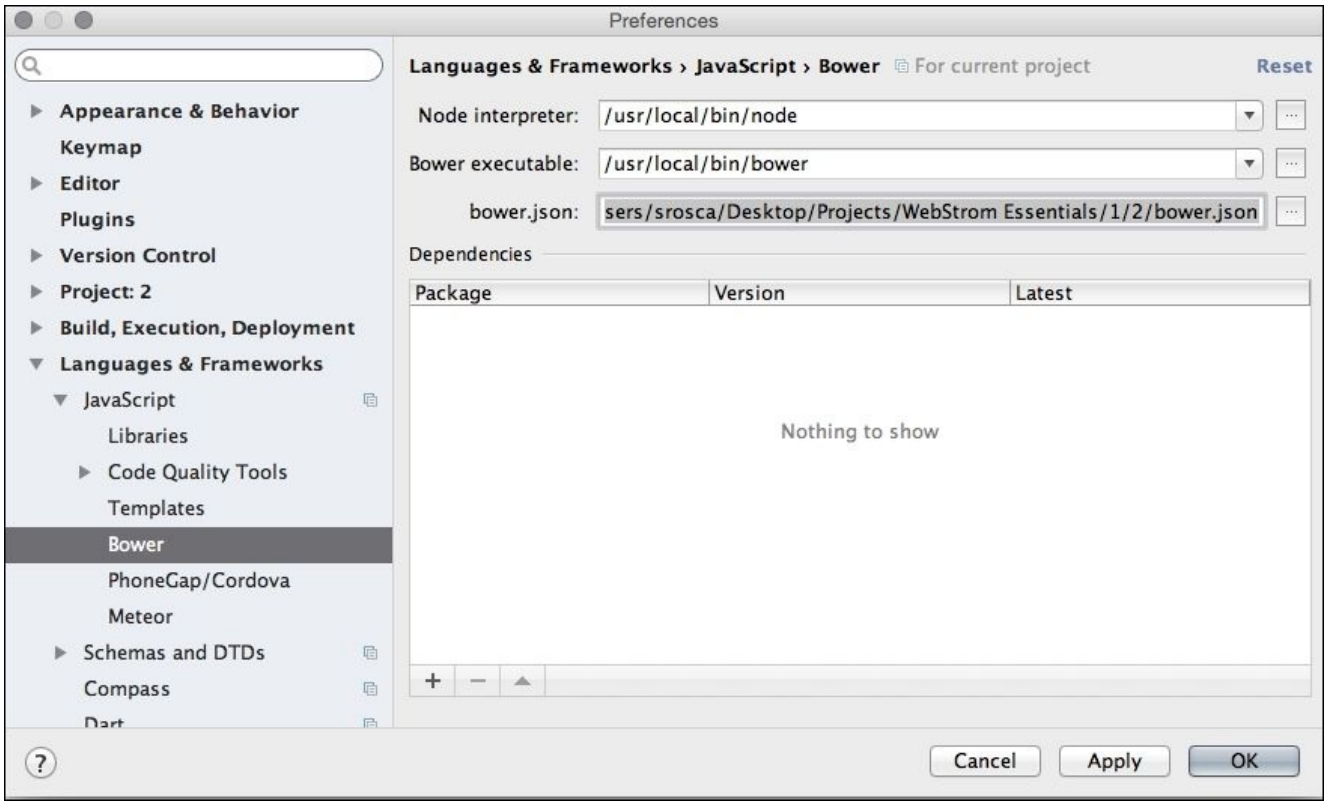
  ### Note

  Please take note that Bower installs the package in the current folder; so before you run the command, be sure that you are in the right folder.

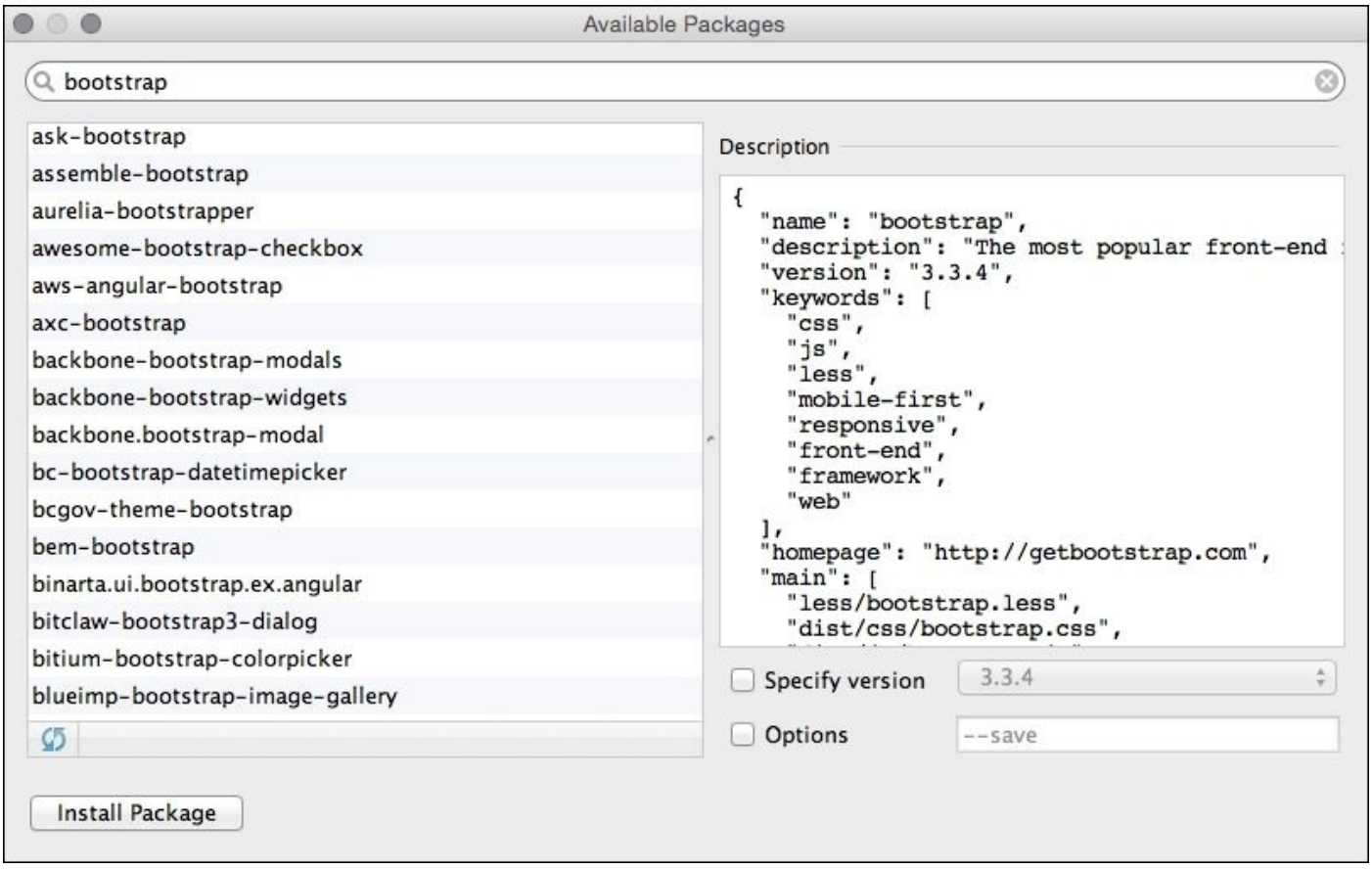
  Bower can use `install` for the following sources as package:

  - Registered package name `bower install bootstrap`
  - GitHub shorthand `bower install twbs/bootstrap`
  - Git endpoint `bower install git://github.com/twbs/bootstrap.git`
  - URL `bower install https://maxcdn.bootstrapcdn.com/bootstrap/3.3.4/css/bootstrap.min.cs`

- The second option is to go to the Bower settings page accessible by **Preferences** | **Languages & Frameworks** | **JavaScript** | **Bower,** and select the install icon + from the **Dependencies** section:



In the new screen that opens up, search and select the package that you want to install, as seen in the following screenshot:



In both cases, the terminal or the install dialog, if you want to save your dependencies or

the development dependency in the `bower.json` file, you have to use the `--save` or `--saveDev` option in either the terminal command, or the option field from the dialog.

Now that you have learned how to work with package managers in WebStorm, you are going to learn how to install and use two of the popular task runners, Grunt and Gulp.

Using Grunt

Grunt is a JavaScript test runner that was built to automate the repetitive tasks that we have to run during development. It can be used for tasks like minification, compilation, linting, unit testing, or any other tasks that your project needs. The Grunt ecosystem has hundreds of plugins that you can choose from.

Before using Grunt, we need to install the CLI globally. This doesn't install Grunt, but it will run the version of Grunt that was installed next to the configuration file, `Gruntfile.js`. In this way, we can have multiple versions of Grunt installed. To install the tool, we are going to use NPM once again, so, run the following command in the terminal window:

```
npm install -g grunt-cli
```

In the next few steps, we are going to add Grunt to the project that we used in the first chapter. So, go ahead and open the project created in the first chapter, or get it from the Git repository at https://github.com/srosca/youAreHere.

With the project open, the first thing we need to do is to initialize NPM so that we can download Grunt and the necessary plugins. In the terminal window, run `npm init` and answer the questions (you can leave them to the default values for the moment). This will create the necessary `package.json` file for saving our future dependencies.

Now we need to install Grunt for the current project; so, run the following command:

```
npm install grunt --save-dev
```

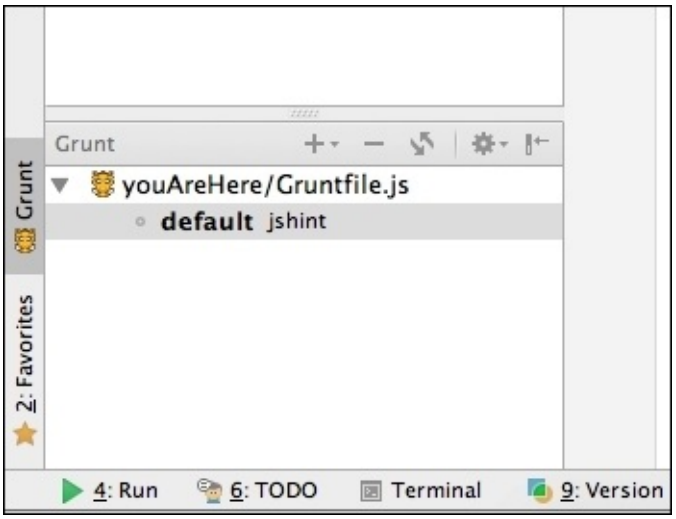
Next we have to create the `Gruntfile.js` configuration file in the root folder of the project, and fill it with the following code:

```
module.exports = function(grunt) {
    grunt.initConfig({
        jshint: {
            files: ['Gruntfile.js', 'js/*.js' ],
            options: {
                globals: {
                    jQuery: true
                }
            }
        }
    });

    grunt.loadNpmTasks('grunt-contrib-jshint');

    grunt.registerTask('default', ['jshint']);

};
```

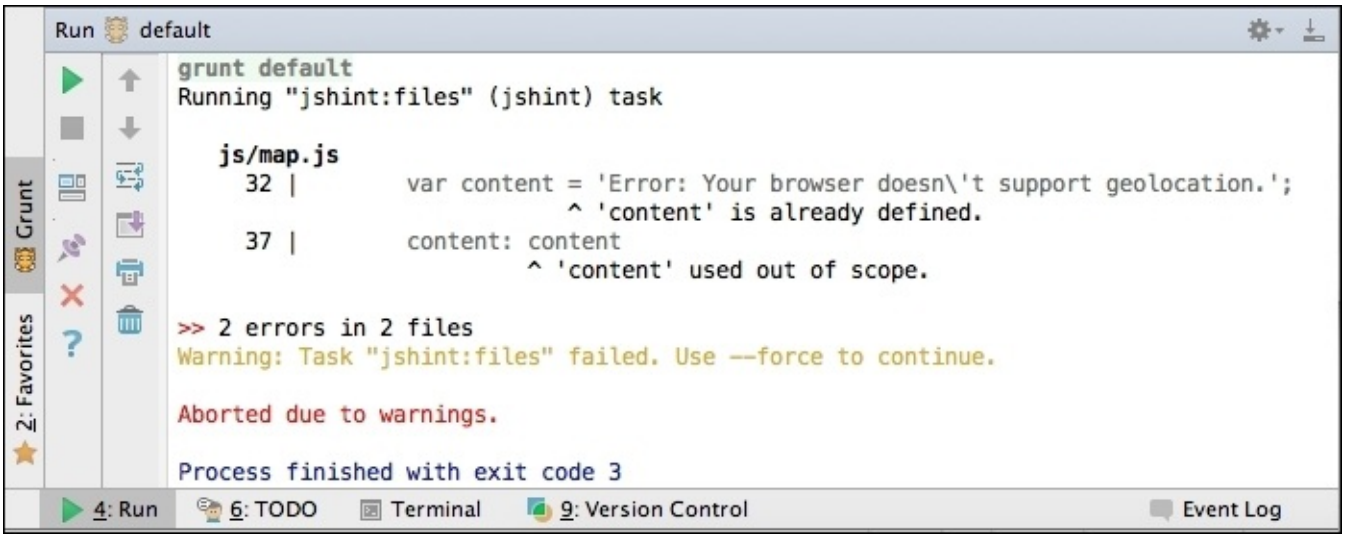
This creates a simple task that does a simple `jshint` on our `.js` code. Before we use this task, we also need to install the `jshint` Grunt plugin. So run the following command in the terminal to install and save it as a development dependency:

```
npm install grunt-contrib-jshint --save-dev
```

Now we can either run the task in the terminal by running Grunt, or use the Grunt tool window. This window can be found in the tools menu or in the side bar:



If we double-click the task or select run from the context menu, the task will start and the output will be displayed in the run window:



As we can see in the preceding screenshot, we already have some errors in the output. So, before we fix them, let's see how we can create a task that watches our files for any changes, and runs the `jshint` when they are changed. For that, we first need to install another Grunt plugin that does that. So in the terminal window, run the following command:

```
npm install grunt-contrib-watch --save-dev
```

After the plugin is installed, open the `Gruntfile.js` configuration file and load the `npm` task after the `jshint` one, by adding the following command:

```
grunt.loadNpmTasks('grunt-contrib-watch');
```

We have to set the `jshint` task force option to make it stop in case of an error, so add the following code to the option sections:

```
force: true,
```

Now after the `jshint` task in the `initConfig` block, add a new task:

```
watch: {
    files: ['<%= jshint.files %>'],
    tasks: ['jshint']
}
```

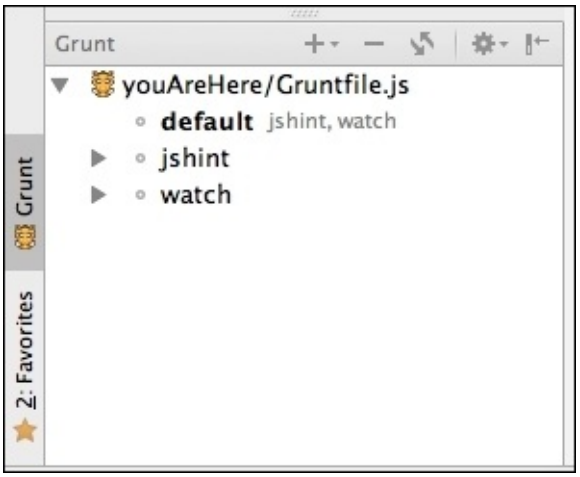
Add the newly created task to the default one so that `jshint` is run before the watch and we can see the results before we change anything:

```
grunt.registerTask('default', ['jshint, watch']);
```

As you can see, we are using the files defined for the `jshint` task. Your final code should now look like this:

```
module.exports = function(grunt) {

    grunt.initConfig({
        jshint: {
            files: ['Gruntfile.js', 'js/*.js' ],
            options: {
                force: true,
                globals: {
                    jQuery: true
                }
            }
        },
        watch: {
            files: ['<%= jshint.files %>'],
            tasks: ['jshint']
        }
    });

    grunt.loadNpmTasks('grunt-contrib-jshint');
    grunt.loadNpmTasks('grunt-contrib-watch');


    grunt.registerTask('default', ['jshint', 'watch']);

};
```

Now in the **Grunt** window, if we click the refresh icon  , we will see that the folder tree has changed, reflecting the newly added task:



If we start the watch task, we will see that now, after running `jshint`, it will wait for changes in the files. So now we can change the `map.js` file to fix the errors by taking the content variable definition outside of the 'if' statement:

```
function handleNoGeolocation(errorFlag) {
    var content;
    if (errorFlag) {
        content = 'Error: The Geolocation service failed.';
    } else {
        content = 'Error: Your browser doesn't support geolocation.';
    }
    var options = {
        map: map,
        position: new google.maps.LatLng(60, 105),
        content: content
    };
    var infowindow = new google.maps.InfoWindow(options);
    map.setCenter(options.position);
}
```

After we make the change, we will see that if we force a save by pressing ⌘ + *S* , Grunt will pick up on the change in file, and will rerun the `jshint` task.

Please note that in WebStorm, the files are saved automatically when we change focus from the current window or by pressing ⌘ + *S*.

In this section, we have automated our task with Grunt and WebStorm. Next, we are going to recreate this flow with Gulp, and see how we can work with it inside WebStorm

Using Gulp

Gulp is another task runner for development tasks. It's similar to Grunt in terms of what it does, but it does that in a different way. It uses streams to link tasks by pasting the output of one to the next one as input; hence, it uses fewer I/O requests. One main advantage of this is that Gulp is a lot faster than Grunt. It also favors the code-over-configuration flavor of specifying the tasks.

To see it in action, we are going to recreate the Grunt flow into Gulp. Before we do that, we need to install it. Open the terminal and install Gulp globally:

```
npm install --g gulp
```

This will install Gulp globally, so we can use it in all projects. We can still use a different Gulp version for projects by installing it locally. Run the following command in the terminal:

```
npm install --save-dev gulp
```

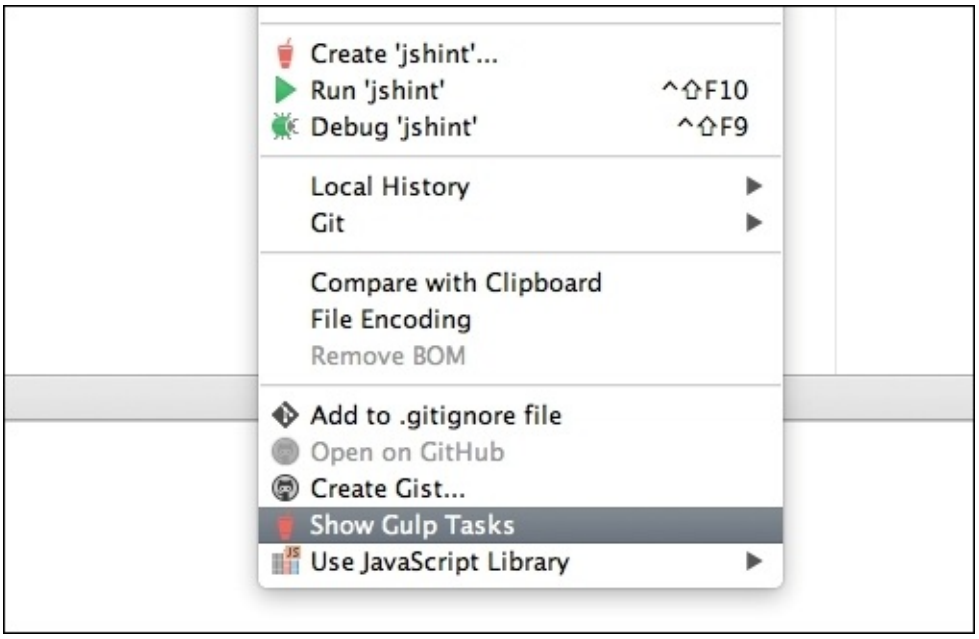
This will install Gulp locally in the project, and save it as a development dependency. In this way, we can have a different version of Gulp for the project that we are working for. Next, we need to install the plugins that we need in our project, So in the terminal, run the next command to install them as a development dependency:

```
npm install gulp-jshint --save-dev
npm install jshint-stylish --save-dev
```

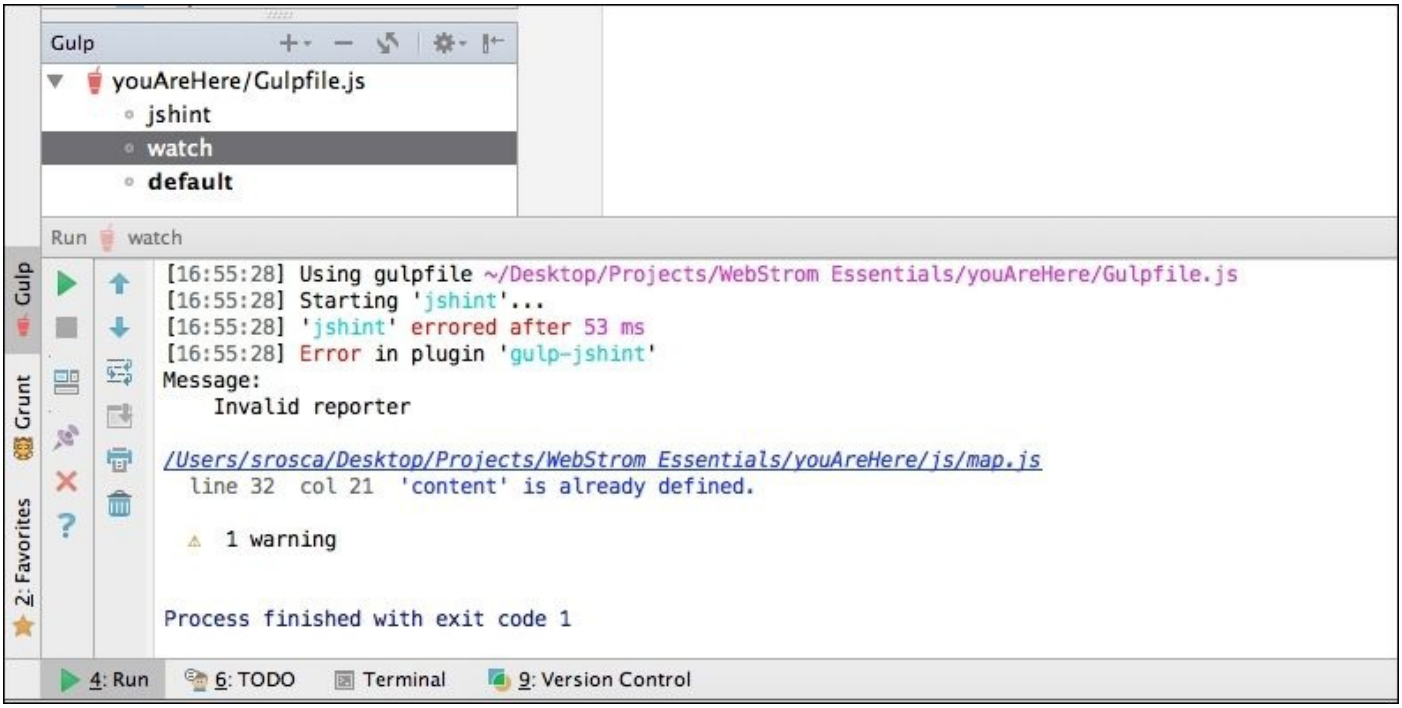
We need to create the `Gulpfile.js` to define the tasks for Gulp; so, go ahead and create it in the root folder. Once you have the file open, create the following content:

```
var gulp = require('gulp');
var jshint = require('gulp-jshint');

gulp.task('jshint', function() {
    return gulp.src('js/*.js')
        .pipe(jshint())
        .pipe(jshint.reporter('jshint-stylish'))
        .pipe(jshint.reporter('success'));
});


gulp.task('watch', ['jshint'], function(event) {
    gulp.watch('js/*.js', ['jshint']);
});
```

As you can see, Gulp uses the code approach when defining a task. With the file open, we can access the Gulp window by selecting **Show Gulp Tasks** from the context menu:



This will open a window similar to Grunt that lists all the tasks available. From this window, we can start the watch tasks that will watch the files for changes and run `jshint` when there is a change. In the next screenshot, you can see the Gulp window and the output of the job:



As you can see, there are some similarities between Gulp and Grunt in the final result, but also some differences in the way you build the tasks.

In this chapter, we have created some simple automated tasks but this can be taken much further. You can create a task for everything that you need to automate in your development process.

Summary

In this chapter, you have learned some of the modern flows of web development package managers and task runners. Now you will be able to quickly add new packages to your project, save them as dependencies, and create tasks for all your repetitive jobs.

In the next chapter, we are going to see how WebStorm can help us build more complex applications with the help of frameworks like AngularJS, React, Express, and Meteor.

Chapter 5. AngularJS, React, Express, and Meteor – Developing Your Web Application

In the previous chapter, we learned the ways in which WebStorm helps us when working with the package manager and build tools, and about the workflow for modern development.

In this chapter, we are going to deal with a couple of the most essential and progressively developing web frameworks and platforms, and then learn how indispensable WebStorm can be when working with them. We will get acquainted with the following technologies in relation to WebStorm:

- AngularJS: a client-only framework
- React: a library for building user interfaces
- Express: a minimalist web framework for Node.js
- Meteor: a full-stack framework running on top of Node.js

AngularJS

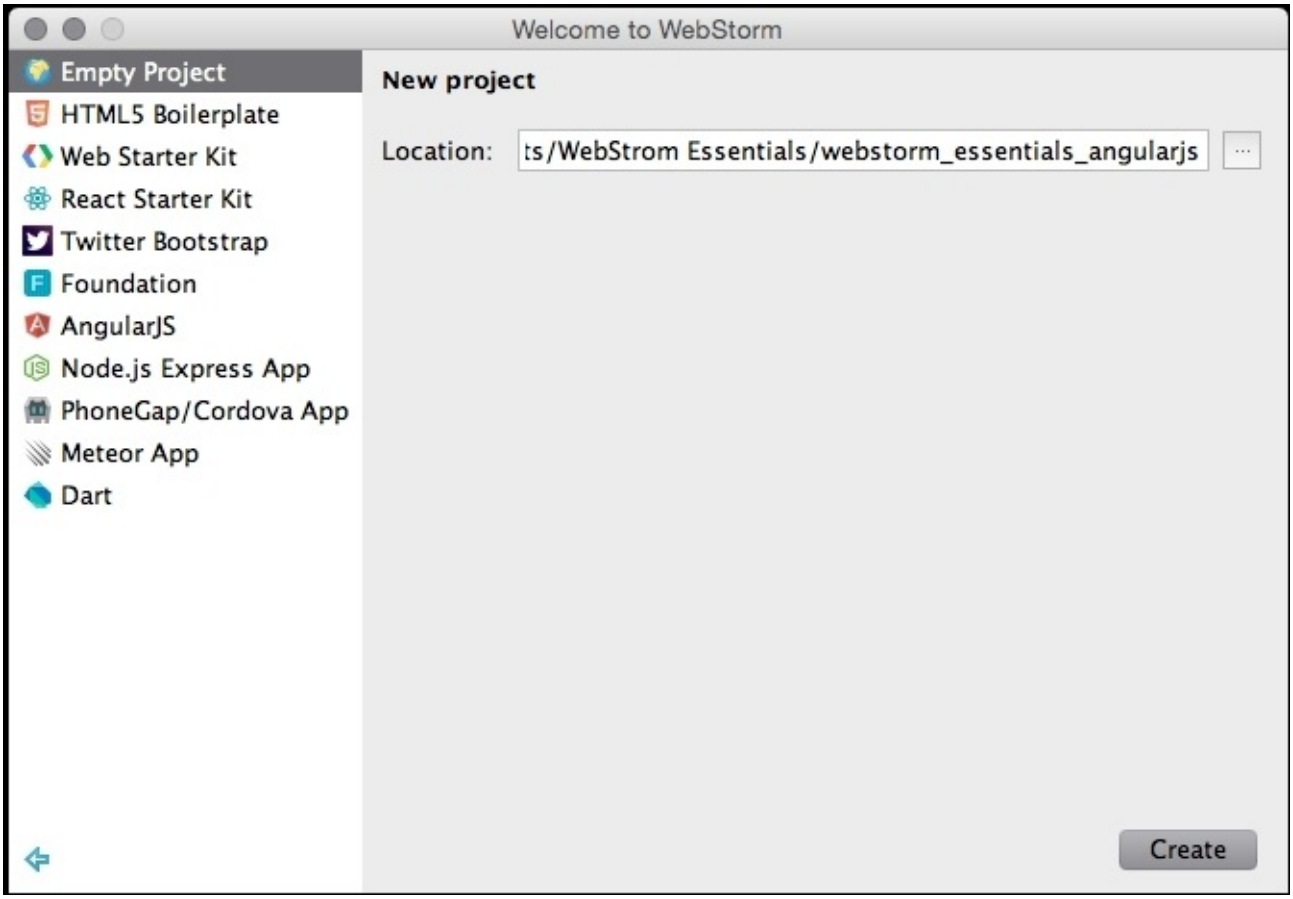
> *"Superheroic JavaScript MVW Framework AngularJS is what HTML would have been, had it been designed for building web-apps. Declarative templates with databinding, MVW, MVVM, MVC, dependency injection and great testability story all implemented with pure client-side JavaScript!"*
>
> —https://angularjs.org

As presented on the developer's website, AngularJS is a client-only framework for building single-page web applications aimed to simplify the development and testing of web applications. In this section, we are we are going to learn about the considerable assistance that WebStorm can provide when using AngularJS, by creating a very simple blog application.

Our application will be a simple blog CMS that will perform the following tasks:

- Display a list of entries
- Display a single entry
- Add a new entry

First, using the steps from the previous chapters, create a new **Empty Project**. When creating a new project, you will notice that there is a project type named **AngularJS**. You can use it in your development activities; it already comprises of all the necessary libraries and setting files. However, for now we are creating a blog from scratch so that you can see the multiple ways in which the various actions can be performed.



Preparing the tools and libraries

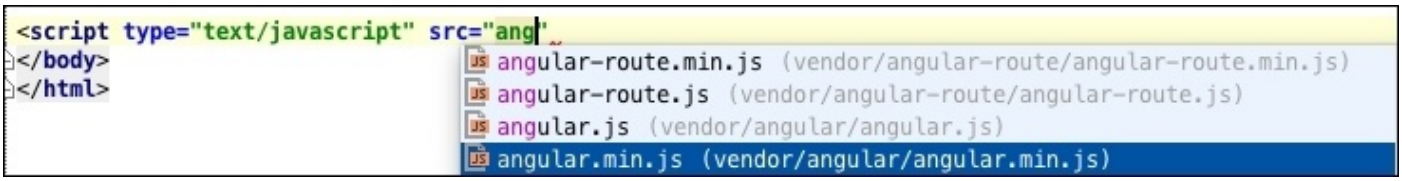
Once we have created our project, we need to install all the dependencies of our app. We will use the AngularJS library, a routing library named Angular Route, and a styling library like Bootstrap. For the installation process, we will use Bower.

I don’t like it when Bower installs the components to the default in `bower_components` directory. So, let us create a `.bowerrc` file and indicate that we want our packages to be installed into the `vendor` directory, as shown in the following screenshot:

Now we can install all the necessary packages. First of all, we need AngularJS. Type and execute the following command in the terminal:

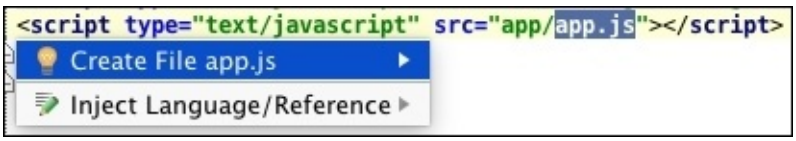
```
bower install angular.
```

This will download AngularJS inside our `vendor` folder, but we still need to include it in the `index.html` file. With this file open at the end of the `body` section, add a `<script>` tag and then start simply typing, for example, `ang`. Use autocomplete to let WebStorm find the necessary `.js` file:



WebStorm will know from now on that our application is based on Angular, and the `ng-` autocomplete and others will work inside your project.

After this, add another `<script>` tag, and in the `src` attribute, type `"app/app.js"`; this will be the core of our application. This file doesn’t exist yet, but you can automatically create it. Select the file name in the `src` attribute and click on *Alt* + *Enter* . Then, click on the **Create File app.js** item, and WebStorm will create the missing file for you in the newly opened tab.



Next, let us install Bootstrap. To do so, you can use Bower from the Terminal window:

```
bower install bootstrap
```

Next, just like we did with angular sources, include the Bootstrap `.css` and `.js` files into your project (using the link and `<script>` tags, respectively). As Bootstrap uses jQuery and this will be installed too, don't forget to add a `<script>` tag for jQuery as well.

Another library that we are going to use in our project is the angular-route library. So, we need to use bower again to install it and then add it to the `index.html` file:

```
bower install angular-route
```

All the manipulations that we have performed so far should result in the following code:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>MyAngularApp</title>
    <link rel="stylesheet"
href="vendor/bootstrap/dist/css/bootstrap.min.css">
    <link rel="stylesheet" href="css/app.css">
</head>


<script type="text/javascript" src="vendor/jquery/dist/jquery.min.js">
</script>
<script type="text/javascript"
src="vendor/bootstrap/dist/js/bootstrap.min.js"></script>
<script type="text/javascript" src="vendor/angular/angular.js"></script>
<script type="text/javascript" src="vendor/angular-route/angular-route.js">
</script>
<script type="text/javascript" src="app/app.js"></script>
</body>
</html>
```

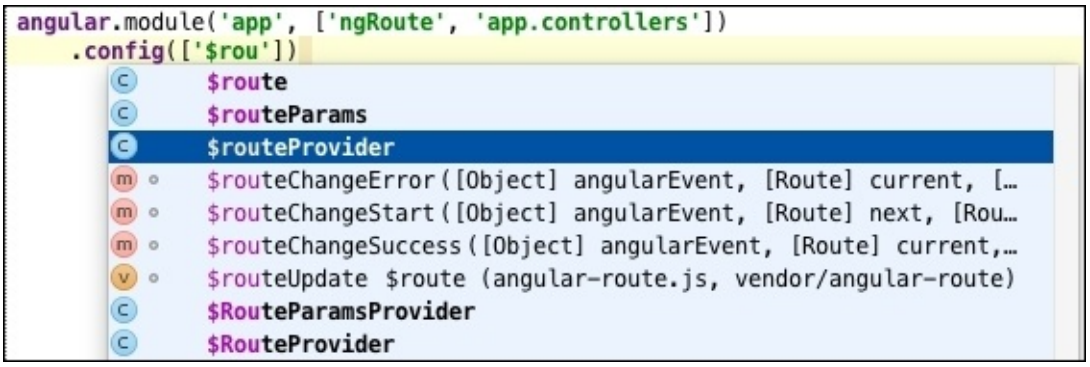
Immersing in AngularJS

Now we can start building our application. Open the `app.js` file, declare a new module called `app`, and then add `ngRoute` as a dependency:

```
angular.module('app', ['ngRoute'])
    .config(function($routeProvider){

        });
    });
```

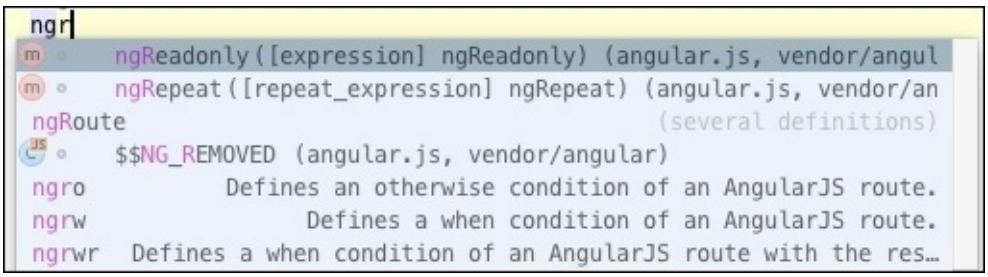
The following screenshot is the output of the preceding code:

You can use autocomplete so that all the code that you write next is automatically suggested by WebStorm. This not only reduces development time by removing the necessity to write everything manually, but also helps you to better realize and decide what is more appropriate in the current context:



However, this is not the only assistance that WebStorm provides you with. WebStorm supplies so-called quick templates, which enable you to quickly write some standard code. For instance, in our case, we will need to indicate three routes—to the entries list, entry page, and the page to add a new entry.

This is performed with the `$routeProvider` construction, which can comprise certain parameters. We need just two, `templateUrl` and controller, inside a `when` condition. So instead of typing this manually, you can simply type `ngrw` and press *Tab*:



This will replace `ngrw` with the following code:

```
$routeProvider.when(' ', {
    templateUrl: '',
    controller: ''
});
```

Moreover, the cursor will be placed directly between the two single quotes right after the word `when`. This may seem trifling but you needn't perform any unnecessary manipulations with your mouse.

Our final code for the `app.js` file should look like the following:

```
angular.module('app', ['ngRoute'])
    .config([function($routeProvider){
        $routeProvider.when('/', {
            templateUrl: 'app/views/entries.html',
            controller: 'EntriesController'
        });
    });
```

Now that the new app module is created, we can bind it to the `index.html` page by adding `ng-app="app"` to the `<body>` tag, of course, by using the autocomplete. The final `<body>` opening tag should read as follows:

```
<body ng-app="app">
```

We should also add a special container where Angular can render our templates. Insert the following code after the opening `<body>` tag:

```
<div class="container app-container">
    <div ng-view></div>
</div>
```

We have created a container `<div>` for styling purposes as well. With this final change, we can move to creating the features of our application.

Loading the initial entries

Our application will have a list of predefined topics that will be displayed on the first load. So, the first thing we need to do is to make sure that the data is loaded in the application. We are not going to use any backend so as not to complicate our application. We will simulate the server response with static JSON files that will be kept in the data folder. For the list of topics, create the `entries.json` file inside the `data` folder and fill it with the following content:

```
[
  {
    "title" : "Hello, WebStorm!", "content" : "Today we are going to speak
about WebStorm and its features"
  },{
    "title" : "Immersing into WebStorm", "content" : "WebStorm provides a
bunch of advanced technologies, which help us increase our development
efficiency"
  }
]
```

Now we need to create the service that stores the data. In the `app.js` file, create a new `app.services` module, and fill it with the following code. We will use a service so that our data is available in multiple controllers and views. You can see that we have already created the `add` method, because we want to be able to create new entries in our blog:

```
angular.module('app.services', [])
    .service('store', function ($http) {
        var store = {
            entries: [],

            add: function (item) {
                this.entries.push(item);
            }
        };

        return store;
    });
```

Next, we need to modify the `app.module` to include the new dependency; so after the `ngRoute`, we need add the new module `app.services`. We also need to create a new run block that will load the initial data when the application is started. The final code should look like this:

```
angular.module('app', ['ngRoute', 'app.services'])
    .config(function ($routeProvider) {
        $routeProvider.when('/', {
            templateUrl: 'app/views/entries.html',
            controller: 'EntriesController'
        });

    }).run(function ($http, store) {
        $http.get('data/entries.json')
            .success(function (data) {
```

```
            store.entries = data;
        });
});
```

Displaying a list of entries

For our list, we will need a controller for the data and a view that will display it. Moreover, since we want to display only a short summary in the main page, we also have to create a filter that creates the summary text.

Let's start with this filter so we can use it later in the view. In the `app.js` file, add the following module for the filters:

```
angular.module('app.filters',[])
    .filter('summary', function($filter) {
        return function(item) {
            return item.split(" ").slice(0,5).join(" ");
        };
    });
```

This filter takes the full content of the entry and displays only the first five words. Actually, this filter is agnostic and it doesn't care what it receives; it will take any string, and return the first five words from it. Again, after we have created this new module, we need to add it as a dependency on the app module.

Now we can move to the first controller, create a new module `app.controllers`, and add the first controller of our application, `EntriesController`. The only thing that this controller does is take the data from the store and pass it to the view. After you create this module, don't forget to add it as a dependency:

```
angular.module('app.controllers', [])
    .controller('EntriesController', function ($scope, store) {
        $scope.entries = store.entries;
    })
```

The last step that we need to implement is the view. Create the `entries.html` file in the `views` folder, and fill in the following code:

```
<article ng-repeat="entry in entries | orderBy:'-'">
    <h1><a href="#/entry/{{ entries.indexOf(entry) }}">{{ entry.title }}
</a></h1>
    <p>
        {{entry.content | summary }}
        <a href="#/entry/{{ entries.indexOf(entry) }}">... (read more)</a>
    </p>
</article>
```

Displaying entry details

Now that we have created the list of entries, we also have to create the page that displays each entry detail. We will again need a controller, a view and, a new route definition.

We will create the new definition after the route definition for the list page. The final code should look like the following:

```
angular.module('app', ['ngRoute', 'app.controllers', 'app.services',
'app.filters'])
    .config(function ($routeProvider) {
        $routeProvider.when('/', {
            templateUrl: 'app/views/entries.html', controller:
'EntriesController'
        }).when('/entry/:index', {
            templateUrl: 'app/views/entry.html', controller:
'EntryController'
        })
    })
```

Now create a new controller in the `app.controllers` module that reads the index from the route parameters and loads the specified entry, as follows:

```
.controller('EntryController', function ($scope, $routeParams, store) {
    $scope.entry = store.entries[$routeParams.index];
})
```

We now have to create the `entry.html` file, and fill it with the view for our entry:

```
<article>
    <h1>{{ entry.title }}</h1>
    <p>{{entry.content}}</p>
    <p><a href="#/">Go back</a></p>
</article>
```

This view displays the entry title and a full content text. With this view, we have finished the entry detail functionality; and, as you can see, you can already navigate between the list and the details pages. The only thing that we have to create now is the page where we add a new entry.

Adding a new entry

Before we even start on this feature, we need to add a menu that will allow us to navigate to this page. So, open the `index.html` file, and add the following code before the container `<div>` of your application. This code will add a header menu with the link to this page. For this menu, we are going to use the familiar Bootstrap syntax from the previous chapters.

```
<nav class="navbar navbar-default navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <a class="navbar-brand" href="#/">My Angular App</a>
        </div>

        <div class="collapse navbar-collapse">
            <ul class="nav navbar-nav">
                <li>
                    <a href="#/new-entry/">Add new</a>
                </li>
            </ul>
        </div>
    </div>
</nav>
```

Now that we have the menu for navigating to the page for adding a new entry, we need to create a new route definition for this. Following the same steps as we did for the previous routes, and add the following definition:

```
.when('/new-entry/', {
    templateUrl: 'app/views/new-entry.html',
    controller: 'NewEntryController'
})
```

Now we can create the controller and the view. Again, in the `app.controllers` module, add the new controller. In this controller, we only need to define two functions: one to add the entry to our blog, and one to clear the form, as follows:

```
.controller('NewEntryController', function ($scope, $location, store) {
    $scope.add = function () {
        store.add($scope.entry);
        $location.path('/');
    };
    $scope.clear = function () {
        $scope.entry = {};
    }
});
```

We can now create the view in the `new-entry.html` file. We will again use the Bootstrap syntax for styling. You might also notice the `ng-model` that binds the inputs with the controller:

```
<form>
    <div class="form-group">
        <label for="title">Title</label>
        <input type="text" class="form-control" id="title" ng-
```

```
model="entry.title">
    </div>
    <div class="form-group">
        <label for="title">Content</label>
        <textarea class="form-control" rows="5" ng-model="entry.content">
</textarea>
    </div>
    <div class="form-group">
        <button type="submit" class="btn btn-default" ng-
click="add()">Save</button>
        <button type="submit" class="btn btn-default" ng-
click="clear()">Clear</button>
    </div>
</form>
```

Now that we have created all the features for our application, we can move to the final step of styling our application, and add the final touches.

Styling the application

One of the first things that we can do is add the footer for our application. In the `index.html` file, add the following after the `app` container `<div>`:

```
<footer class="footer copy">
    <div class="container">
        <p class="text-muted pull-right">Copyright © 2015. All rights
reserved</p>
    </div>
</footer>
```

You may remember that we have included the `app.css` file in our `index.html`; so, go ahead, create this file in the `css` folder, and add the following code:

```
.app-container{
    margin-top: 100px
}
```

Once again, since we have used the Bootstrap syntax and the helper classes, the amount of styling code that we have to write is minimal.

With these final steps, we have finished our blogging application. As you just saw, it is very easy to create a complex application with the help of WebStorm and Angular. One thing to keep in mind is that this application is just an example, and there are a lot of improvements and changes that we can make; but that is well beyond the scope of this book.

Next we are going to recreate the same application with the same features in another advanced JavaScript framework: React.

React

> *“A JavaScript library for building user interfaces”*
>
> —http://facebook.github.io/react/

React is a UI framework developed by Facebook that has a different approach to building web applications. It uses a virtual DOM to give the developers a simpler programming model and better performance. Moreover, the data flow in React applications is a one-way reactive, which reduces the Boilerplate and makes it easier to reason about the state of the application.

Our goal for this section is to recreate the blog application that we have created previously, but using React as the UI framework. We will keep the same features that we had in the previous section.

The first step that we need to take is to create a new **Empty Project** for our application. Next, we need to install all the dependencies. Using the steps described earlier in the chapter, create the `.bowerrc` file and the `vendor` folder. Then install `react`, `react-router`, and `bootstrap`. You can do that by running the following single command:

```
bower install  react react-router bootstrap
```

Now that we have all the dependencies installed, we have to include them in the `index.html` file. Again, use the steps previously described, and include the navigation and footer that we did in the Angular example. The final code should look like the following:

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>My React App</title>
    <link rel="stylesheet"
href="vendor/bootstrap/dist/css/bootstrap.min.css">
    <link rel="stylesheet" href="css/app.css">
</head>
<body>
<nav class="navbar navbar-default navbar-fixed-top">
    <div class="container">
        <div class="navbar-header">
            <a class="navbar-brand" href="#/">My React App</a>
        </div>

        <div class="collapse navbar-collapse">
            <ul class="nav navbar-nav">
                <li>
                    <a href="#/new-entry/">Add new</a>
                </li>
            </ul>
        </div>
    </div>
</nav>
```

```
<div class="container app-container">
    <div id="app"></div>
</div>

<footer class="footer copy">
    <div class="container">
        <p class="text-muted pull-right">Copyright © 2015. All rights
reserved</p>
    </div>
</footer>

<script type="text/javascript" src="vendor/jquery/dist/jquery.min.js">
</script>
]<script type="text/javascript"
src="vendor/bootstrap/dist/js/bootstrap.min.js"></script>
<script type="text/javascript" src="vendor/react/react.js"></script>
<script type="text/javascript" src="vendor/react-
router/build/umd/ReactRouter.js"></script>

<script type="text/javascript" src="vendor/react/JSXTransformer.js">
</script>
<script type="text/jsx" src="app/app.jsx"></script>
</body>
</html>
```

One thing that might seem out of place in this code is the `text/jsx` type used for the `app` file. React uses a special syntax for its files called JSX. This syntax is similar to XML and it help us write the markup inside the component files.

So now that we have our `index.html` file ready, we can create the `app.jsx` file in the `app` folder, and start filling in the code for our application.

The fist thing that we have to add is the supporting code for our router. To keep our application simpler, we are not going to use any module system. But keep in mind that this is a not good as a regular practice:

```
window.React = React;
var Router = window.ReactRouter;
var Route = Router.Route;
var DefaultRoute = Router.DefaultRoute;
var RouteHandler = Router.RouteHandler;
```

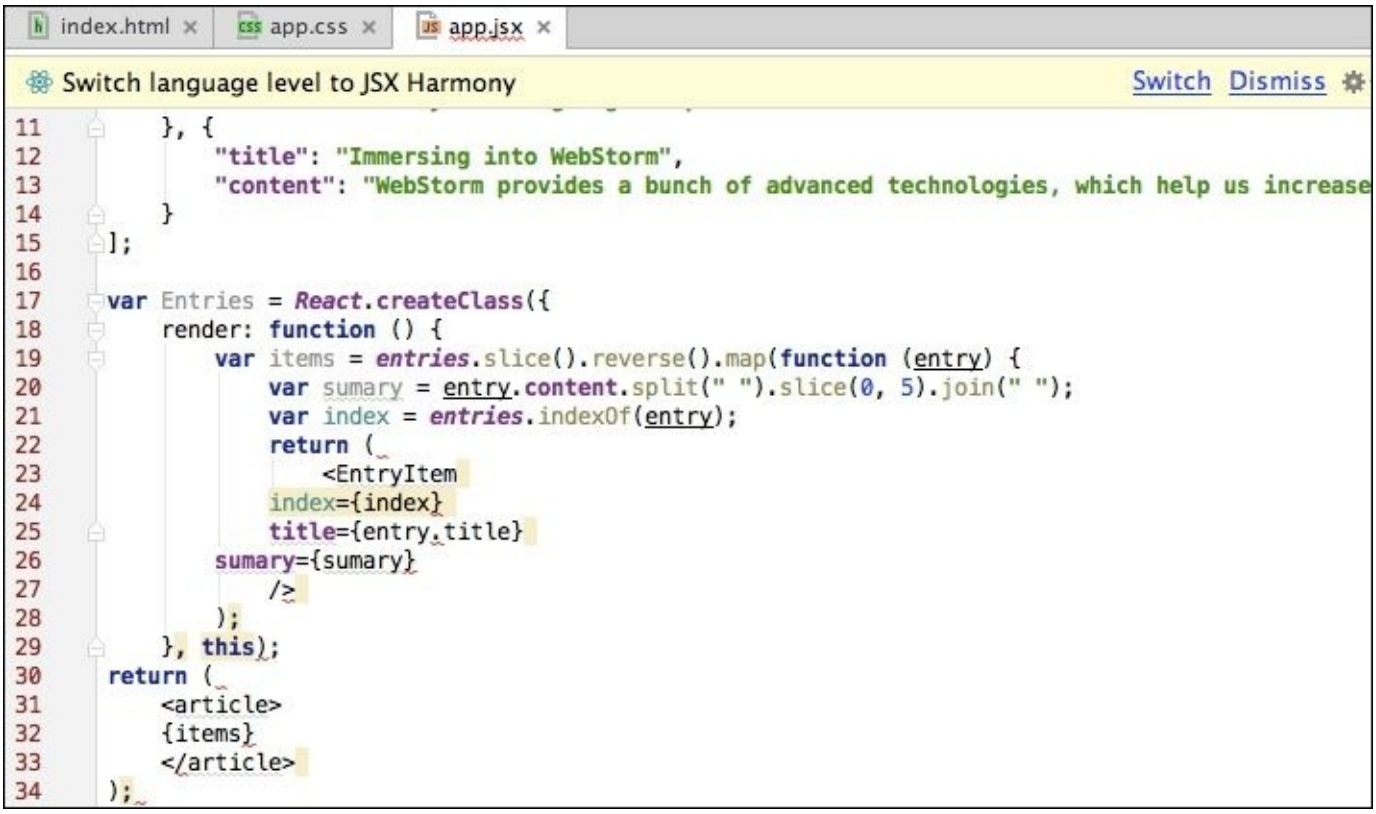
To keep things simple, we will skip the loading of entries from an external JSON file and, instead, create a variable to keep our initial data.

```
var entries = [
    {
        "title": "Hello, WebStorm!",
        "content": "Today we are going to speak about WebStorm and its
features"
    }, {
        "title": "Immersing into WebStorm", "content": "WebStorm provides a
bunch of advanced technologies, which help us increase our development
efficiency"}
];
```

We can now move to creating the features of our application. The first one will be the entries list. As you can see, in React, all the functionality, including the template, is created in JavaScript with the JSX syntax. It might seem odd at first, but later on, it will prove to be a very efficient system for keeping all things together.

```
var Entries = React.createClass({
    render: function () {
        var items = entries.slice().reverse().map(function (entry) {
            var sumary = entry.content.split(" ").slice(0, 5).join(" ");
            var index = entries.indexOf(entry);
            return (
                <EntryItem
                    index={index}
                    title={entry.title}
                    sumary={sumary}
                    />
            );
        }, this);
        return (
            <article>
                {items}
            </article>
        );
    }
});
```

At some point, as WebStorm analyzes your code, it will see that you are using the JSX syntax and prompt you to switch the language level to JSX Harmony. So make sure you click **Switch** at this prompt, or go to the **Preferences | Languages & Frameworks | JavaScript**, and change the language version there:



The `Entries` class relies on the `EntryItem` that we need to create. It is very common in React to compose elements using multiple child elements. You can see that the data is passed from the parent to the child as properties on the props object in the following code:

```
var EntryItem = React.createClass({
    render: function () {
        return (
            <div>
                <h1><a href={'#/entry/' + this.props.index}>
{this.props.title}</a></h1>
                <p>
                    {this.props.sumary}
                    <a href={'#/entry/' + this.props.index}>... (read more)
</a>
                </p>
            </div>
        );
    }
});
```

With this class, we have completed the `Entry` items feature, so we can now move to create the `Entry` detail features. Again, this will just be a simple render method that will display the details for a specific entry. The ID of the entry will be read from the route as a property on the `pros.params` object:

```
var Entry = React.createClass({
    render: function () {
        var id = this.props.params.id;
        var entry = entries[id];
        return(
            <article>
                <h1>{ entry.title }</h1>
                <p>{entry.content}</p>
                <p><a href="#/">Go back</a></p>
            </article>
        );
    }
});
```

The last feature that we need to create is the new entry page feature. For this, we will also need to create two special methods that will handle the save and clear functionality. Another thing worth mentioning in this class is the `Router.Navigation` mixin that we are using. Mixins bring functionality from other classes into our own classes. In our example, we need to be able to navigate programmatically from within the submit method.

```
var NewEntry = React.createClass({
    mixins: [Router.Navigation],

    handleSubmit: function () {
        var title = this.refs.title.getDOMNode().value,
            content = this.refs.content.getDOMNode().value;

        entries.push({
```

```
            title: title,
            content: content
        });
        this.transitionTo('/');
    },
    handleClear: function () {
        this.refs.title.getDOMNode().value = '';
        this.refs.content.getDOMNode().value = '';
    },
    render: function () {

        return (
            <form>
                <div className="form-group">
                    <label htmlFor="title">Title</label>
                    <input type="text" className="form-control" id="title"
                    ref="title"/>
                </div>


                <div className="form-group">
                    <label htmlFor="content">Content</label>
                    <textarea className="form-control" rows="5"
ref="content"></textarea>
                </div>

                <div className="form-group">
                    <button type="submit" className="btn btn-default"
                    onClick={this.handleSubmit}>Save</button>
                    <button type="submit" className="btn btn-default"
                    onClick={this.handleClear}>Clear</button>
                </div>
            </form>
        );
    }
});
```

As you might have noticed, WebStorm is really useful when writing the JSX syntax, since it allows for code completion and checking in a way similar to writing plain HTML or JavaScript code.

The last component class that we need to create is the main `App`:

```
var App = React.createClass({
    render: function () {
        return (
            <div>
                <RouteHandler/>
            </div>
        )
    }
});
```

So far we have created all the components for our application; now we can move to create the route definitions for the features of our application: the entries list (this will also be the default list), the entry details, and the new entry page:

```
var routes = (
    <Route path="/" handler={App}>
        <DefaultRoute  handler={Entries}/>
        <Route path="/entry/:id" handler={Entry}/>
        <Route path="/new-entry/" handler={NewEntry}/>
    </Route>
);
```

The last step before we can test our application is to wire up the router, which is done as follows:

```
Router.run(routes, Router.HashLocation, function (Root) {
    React.render(<Root />, document.getElementById('app'));
});
```

We are now ready to test our application. So, we will go to the `index.html` file and click the browser icon to run the application. As you can see, the functionality is the same as in the Angular application.

I must point out again that the code that we have created here is just for the functionality and features of WebStorm, when working with the React framework; it is not optimized and it doesn’t respect all the best practices when working with this framework.

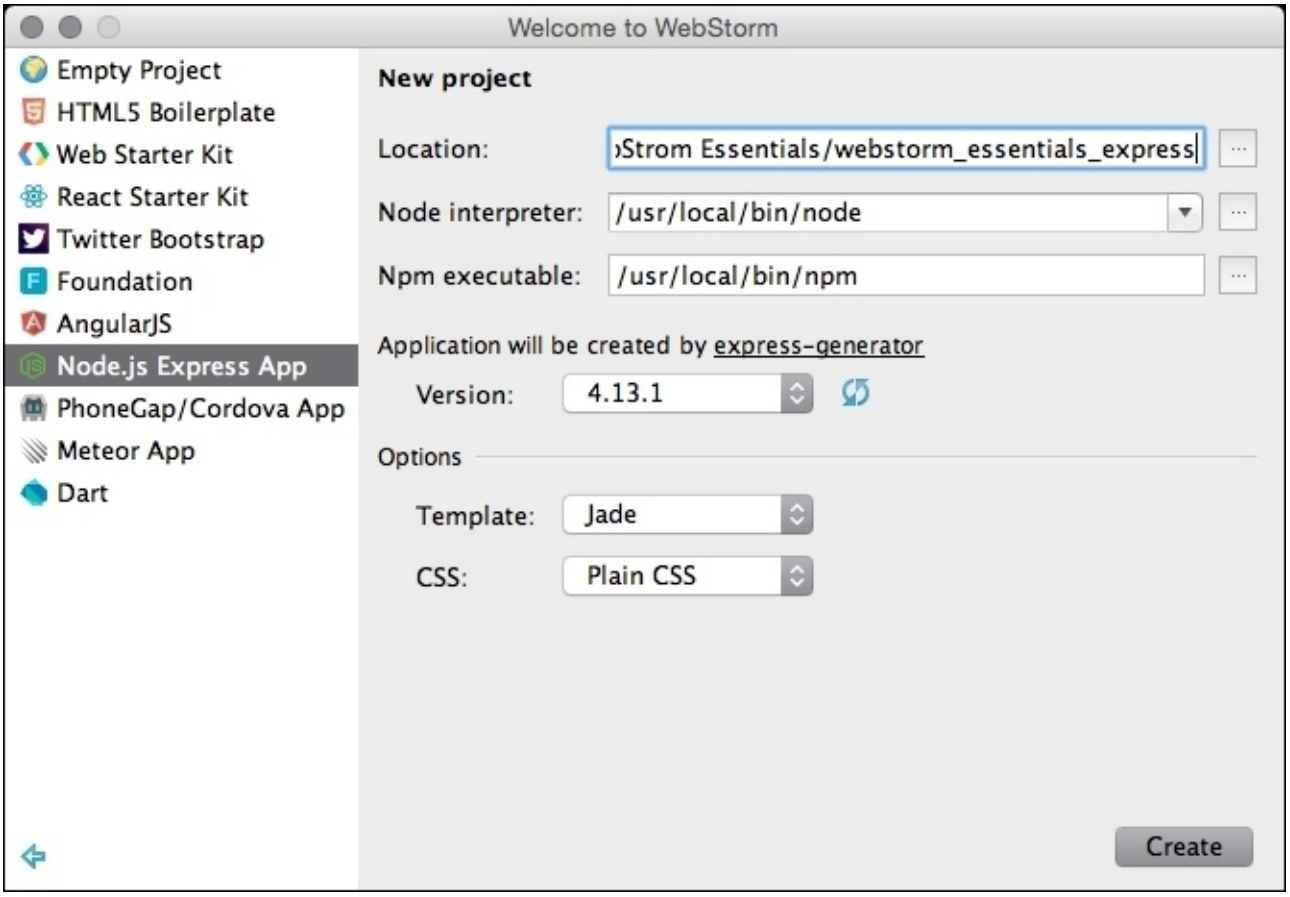
Express

"Fast, unopinionated, minimalist web framework for Node.js"

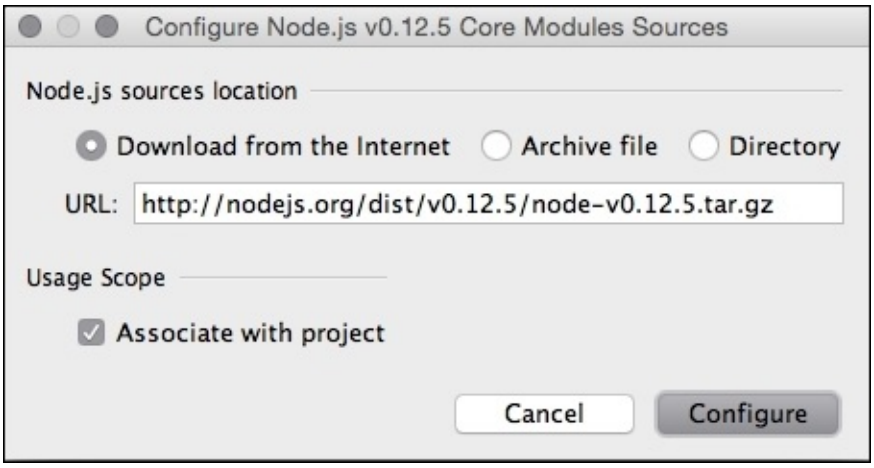
—http://expressjs.com/

In this section, we are going to look at building some apps using JavaScript in the server. We are first going to look at one of the most popular framework builds on top of Node.js, that is, Express. This is one of the most popular frameworks that provides a robust set of features for both, web and mobile applications.

To start with, we need to create a new project based on the **Node.js Express App** template:



WebStorm will create the project, and install all the required dependencies. At some point, you will also be prompted to download and install the Node.js core module sources. This will be required for providing autocompletion and analysis for the Node.js core methods, so make sure you select Configure.



After the completion of all these steps, WebStorm will create a Run/Debug configuration, so you can quickly preview your project. If you select **'bin/www'** to be executed from the **Run** menu or the toolbar, your application will start, and you can preview it in your browser at the address: `http://localhost:3000/`.

As you can see, it is easy to start a Node.js/Express project, including all the dependencies and configurations, with the help of WebStorm. We will not go any deeper in this chapter, but this project is an excellent foundation for any Express application.

Meteor

Meteor is an open source, real-time platform for building web applications. It runs on top of Node.js and uses MongoDB. The framework is rather young—Meteor 1.0 came out, not very long ago, on the 28 October, 2014—but it has already proved itself to be a powerful tool. WebStorm added support for Meteor in version 9, so you can utilize all the techniques that WebStorm suggests.

Before you start building a Meteor application, you need to have Meteor installed on your machine. Linux and OS X, being command line-centric systems, enable you to install Meteor in one click with following command:

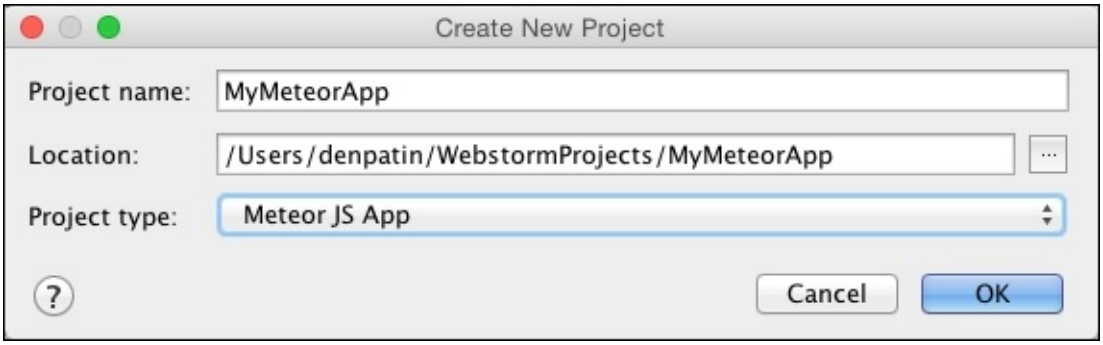
```
curl https://install.meteor.com/ | sh
```

Windows users will need to visit http://win.meteor.com/ and download the `.exe` file, which is stable, although not the latest version. Alternatively, you may check the website, https://github.com/meteor/meteor/wiki/Preview-of-Meteor-on-Windows and try an RC version.

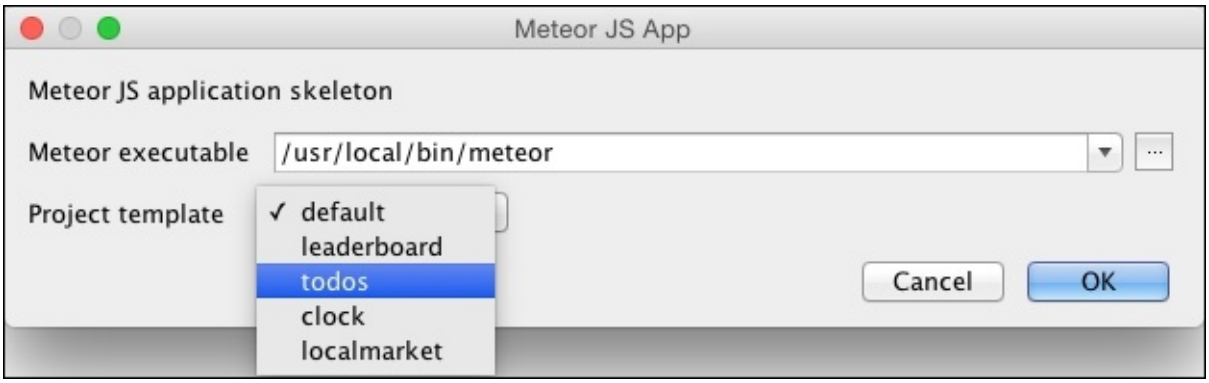
Setting up a new project

Our Meteor environment is ready, so let us move from words to deeds and create a small application to observe where and how WebStorm can assist us in using Meteor.

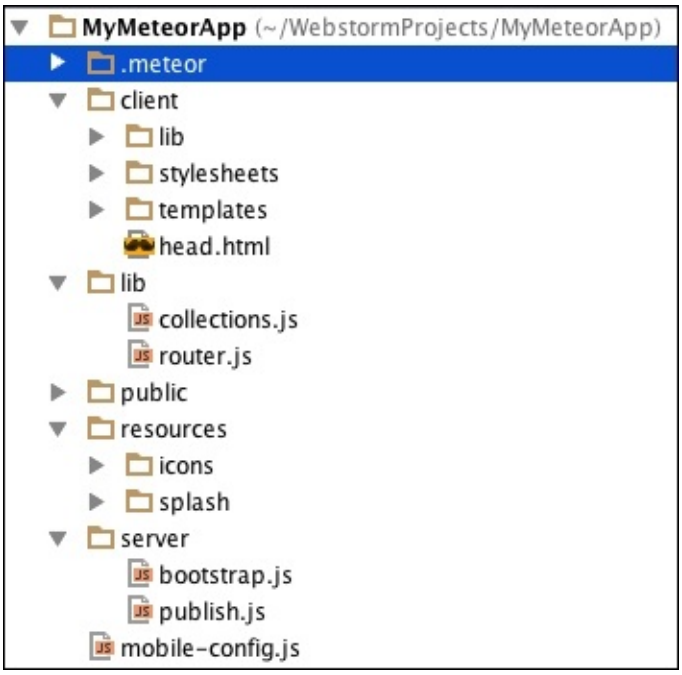
Let us first create a new project called `MyMeteorApp`, as shown in the following screenshot:



In the next window, you will be prompted to select the location of the Meteor executable. WebStorm first tries to locate it automatically, and then, if you need to specify another location, you can do it here. Also, you can use several templates, which are applications that have already been implemented. Let us examine how WebStorm handles Meteor with the help of of the **todos** application example. This is a simple application for tracking what is to be done and what has already been done:

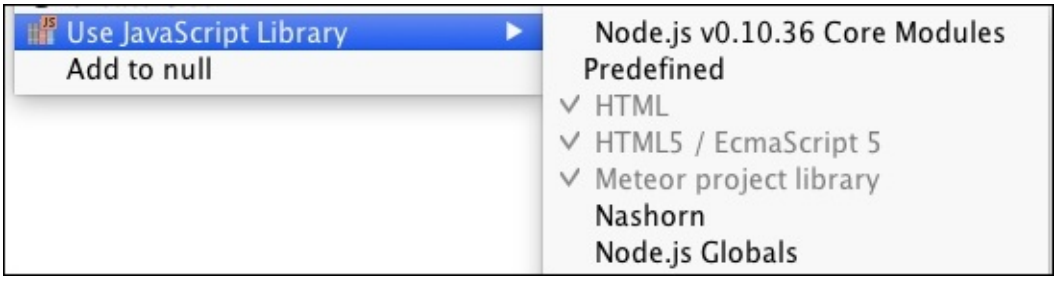


WebStorm needs some time for performing a couple of preparations, such as initializing a new application; then you will see the typical structure of a Meteor project, as seen in the following screenshot:

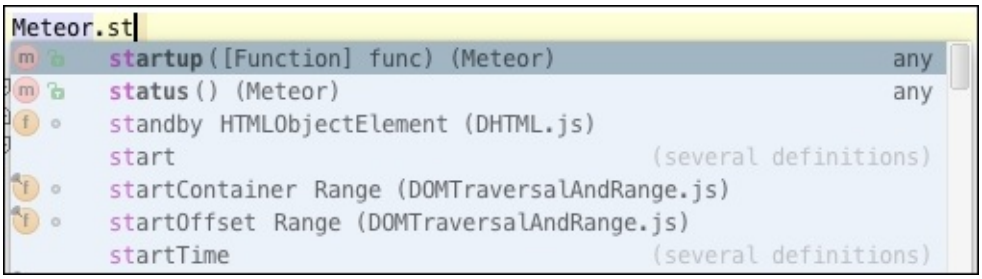


For technical purposes, any JetBrains' IDEA-based IDE creates a special directory called `.idea`, for its own use. The IDE contains information related to the current project, such as the list of all the files, configurations, and even the cursor positions in each of the open files. So, after generating the Meteor project, WebStorm will need some more time to analyze its structure. The nuance in case of Meteor applications is that Meteor has a directory for itself, called—`.meteor`, where it stores information like details about the required packages and their versions. This directory is not controlled by WebStorm, because the contents of this directory can change very often and even grow to a very large size. This can badly affect WebStorm's efficiency. Therefore, the `.meteor` directory is usually under the control of a VCS.

The application is now ready to be opened and run. If you open any `.js` file, you can see that WebStorm understands the Meteor-specific JavaScript and highlights it appropriately. This is because WebStorm automatically uses the Meteor library. You can check this by right-clicking on **Use JavaScript Library** and selecting **Meteor project library**.



This library not only highlights the syntax, but also provides you with a full-fledged autocomplete. If you start typing (for example, `Meteor.st`), you will see that WebStorm, as usual, performs an internal analysis of the content and suggests the most appropriate methods to use in each case:

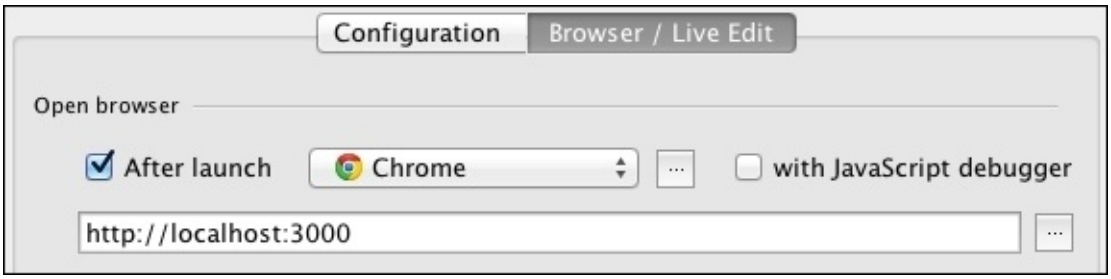


As we are not creating an application from scratch and it is already implemented, we can run it. In a standard case, to run the Meteor applications, you just need to use the command-line interface, and type the `meteor` command. What can WebStorm suggest for this if we can simply use the Terminal?

WebStorm provides a very sophisticated, yet convenient, interface for customizing the run and debug configurations to automate the most frequently performed actions. It is especially useful when you have a lot of parameters that you want to run your application with. So, you won't need to indicate them in the Terminal every time you need them; you will just use the preconfigured cases.

To add a new usage case, go to **Run** | **Edit Configurations…**. This will open a **Run/Debug Configurations** dialog. Click on the plus button, and choose the technology to which you are going to add a usage case:

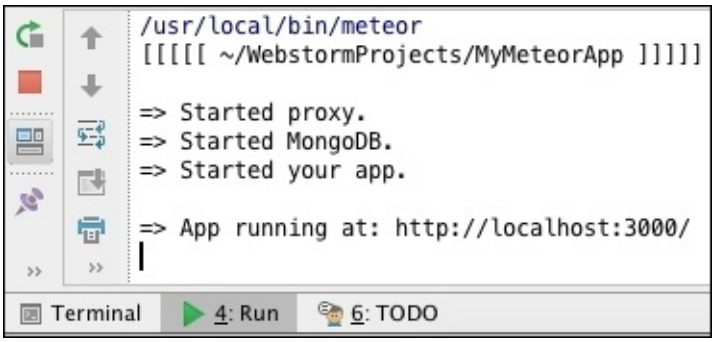
You will see two tabs. On the **Configuration** tab, you can specify the Meteor version to execute your application, program arguments, and environment variables (if needed), and so on. On the **Browser / Live Edit** tab, you are provided with secondary yet useful features, such as the automatic launching of a specific browser when you are running the application:



When you finish setting up all the parameters that you need for a certain case, you can run your application, as follows:

1. Press the green **Play** button on the **Menu** panel of WebStorm (or the green bug button for debugging).
2. Go to **Run** | **Run**, and then choose the configuration that you are going to use.
3. Go to **Run** and run `MyMeteorApp`, which will instantly run your application with the last used configuration.
4. Alternatively, you can just press *Ctrl* + *R* on Mac or *Shift* + *F10* on Windows.

Your application is running! You can see the console output in the **Run** box, as seen in the following screenshot:



Summary

In this chapter, you learned how WebStorm can speed up the development process with the most advanced and popular JavaScript frameworks and platforms. We implemented a mini-blog in both AngularJS and React, and tested the usage of predefined application skeletons for Express and Meteor.

In the next chapter, we will immerse ourselves in the world of mobile development using JavaScript and HTML5.

Chapter 6. Immersing Yourself in Mobile App Development

In the previous chapter, we went through some of the essential web frameworks, and learned how WebStorm can help when working with them. We have seen how JavaScript can be used on both the client and the server.

In this chapter, we are going to learn how to use JavaScript, HTML, and CSS to develop mobile applications, and for setting up our environment to test mobile applications. For this, we are going to focus on the following, which are some of the most-used frameworks of the moment:

- Apache Cordova
- PhoneGap
- Ionic

The main advantage of using the HTML, CSS, and JavaScript frameworks for mobile development is that we can use a single codebase and a single language for all the platforms that we want to target. This can help a lot in terms of the amount of code you have to write, the technologies that you have to learn, and the speed of development.

By the end of this chapter, you will be able to quickly start a mobile project, and test it across multiple devices.

Setting up your system for mobile development

Before we dive into the mobile development world, we need to prepare our system so that we can test the applications we create. For that purpose, we are going to set up some emulators and the necessary tools.

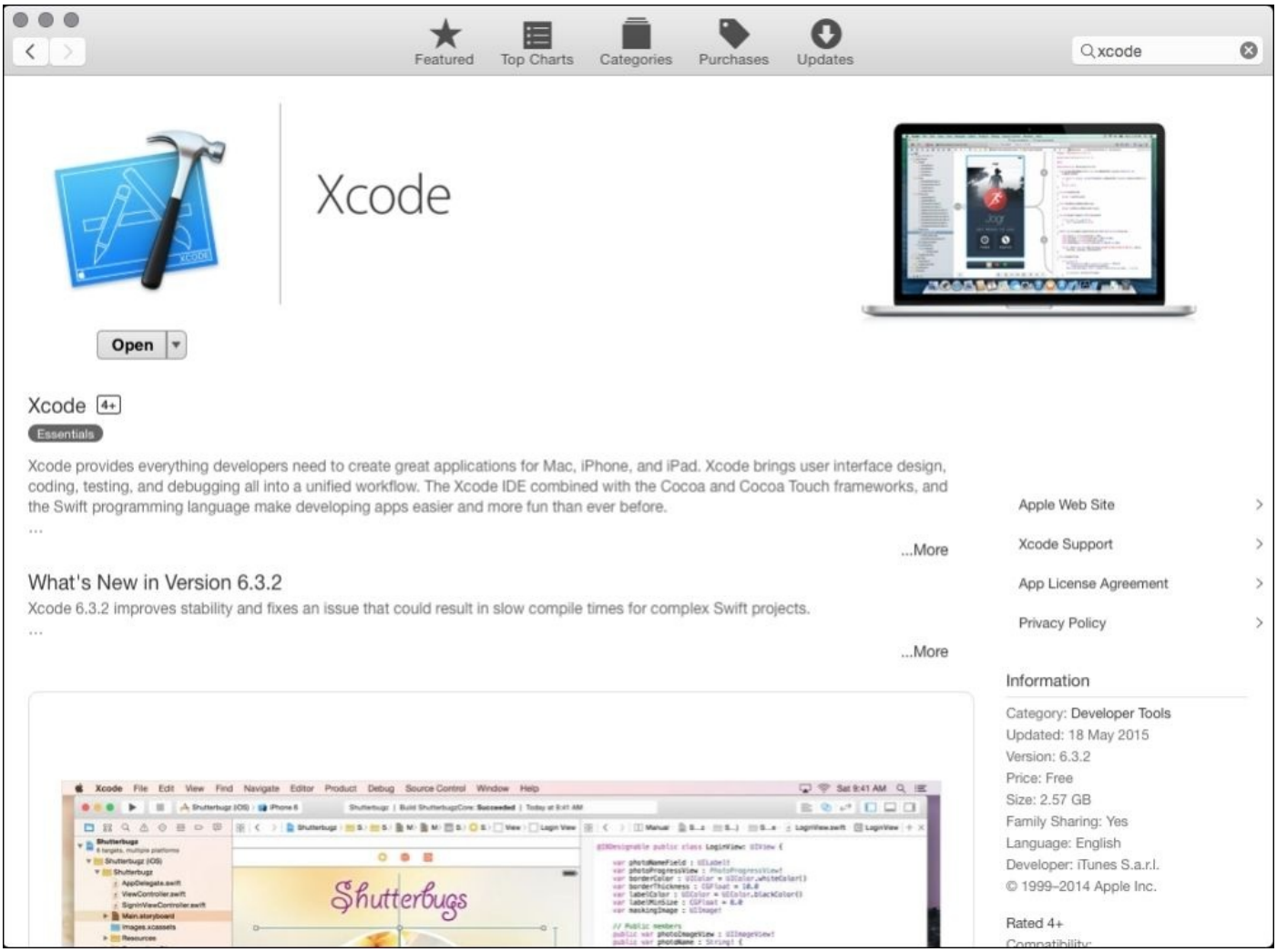
The iOS platform guide

In this section, we are going to set up our environment to deploy apps for iOS devices such as iPhone and iPad. The Apple® tools, which are required to build the iOS applications, run only on the OS X operating system on Intel-based Macs.

Installing Xcode and the SDK

The first tool that you need to install is Xcode. This tool runs only on OS X version 10.9 (Mavericks) or higher, and includes the iOS 8 SDK. If you plan to test your app on an actual device, it must have at least iOS 6.x installed, and you must also be a member of the Apple's iOS Developer Program. In this book, you are going to learn how to deploy the apps to the emulator, for which you don't need to register with the developer program.

You can install Xcode from the AppStore. Search for Xcode and select install:



Once the installation of Xcode is over, we need to install the `ios-sim` tool. This tool is a command-line utility that launches an iOS application in the simulator.

To install `ios-sim`, we need to open a terminal window, and run the following command:

```
npm install -g ios-sim
```

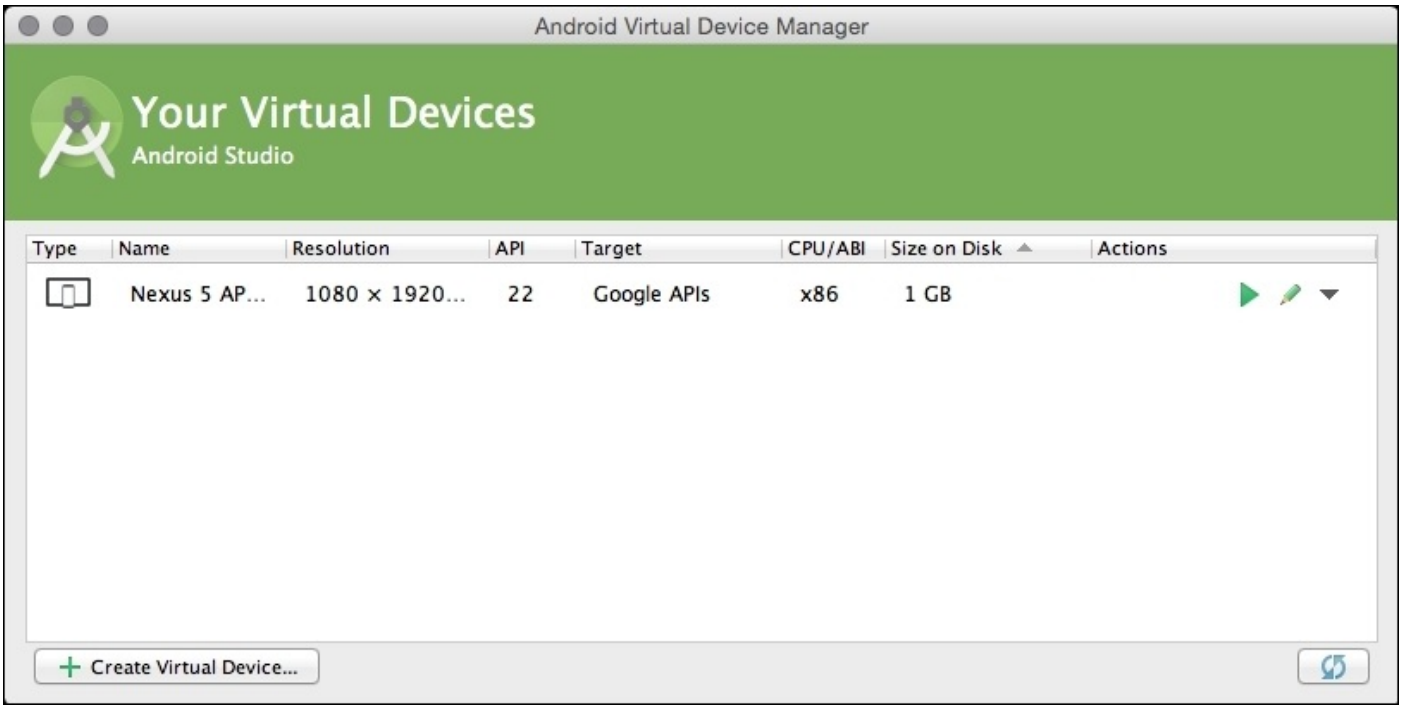
Once the installation of `ios-sim` is done, we have all the tools needed to test our applications on the iOS platform.

The Android platform guide

Now that we have the iOS platform set up, we are going to show you how to set up the environment for the Android apps. The simplest way to get all the required tools is to install Android Studio, the official Android IDE. This is an IDE built on another popular JetBrains product called IntelliJ IDEA Community Edition. It is available for the Windows, Mac OS X, and Linux platforms.

To install Android Studio, download the latest version from http://developer.android.com/sdk/index.html, and then follow the screen instructions.

By default, Android Studio installs the image emulator for Nexus 5. If you want to test your application on other devices, you need to install them in the Android Virtual Device Manager. You can find that in Android Studio under **Tools** | **Android** | **AVD Manager**:



All the frameworks covered in this chapter also support deployment to other platforms besides iOS and Android. So if you want to develop an application for those, you will have to install the necessary SDK or tools. We will not cover the steps for other platforms in this book, but you can find more information by searching for the SDK of each platform.

Some of the other supported platforms are as follows:

- Amazon Fire OS
- BlackBerry
- Firefox OS
- Ubuntu
- Windows Phone
- Tiezen

Now that we have set up the environment for the Android and iOS platforms, we can start building the mobile apps.

Cordova

Apache Cordova is a platform for building native mobile applications using HTML, CSS and JavaScript

—https://cordova.apache.org/

Cordova is a set of JavaScript APIs that allows access to the native functions of the device such as the accelerometer, camera, or geolocation.

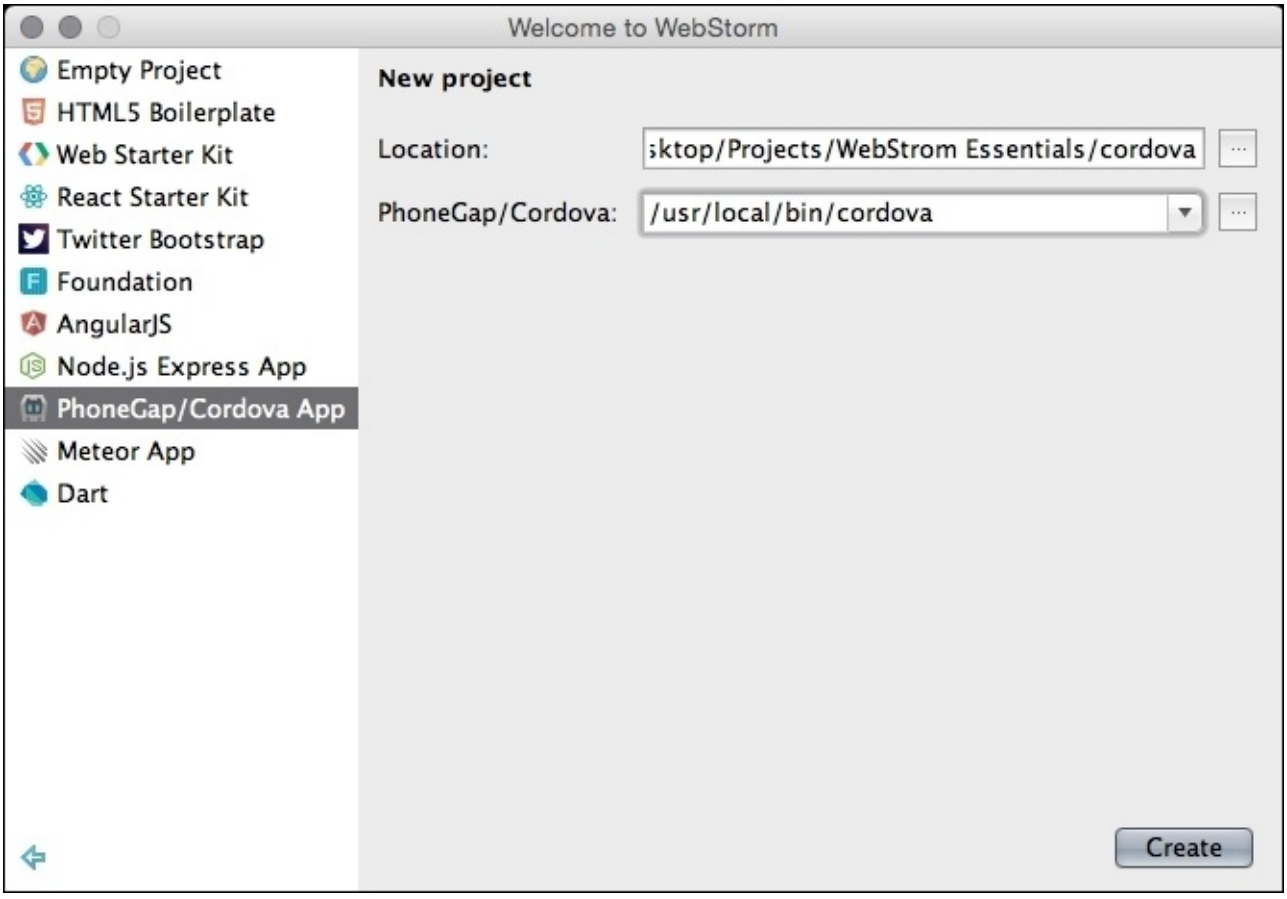
The developer, while writing an app with Apache Cordova, uses web technologies that are packaged using the SDK platform as the native apps. At their core, the apps use HTML5 and CSS3 for the rendering part and JavaScript for the logic.

Before we start building applications with this framework, we need to install it. To do that, we need to run the following command in a terminal window:

```
npm install -g cordova
```

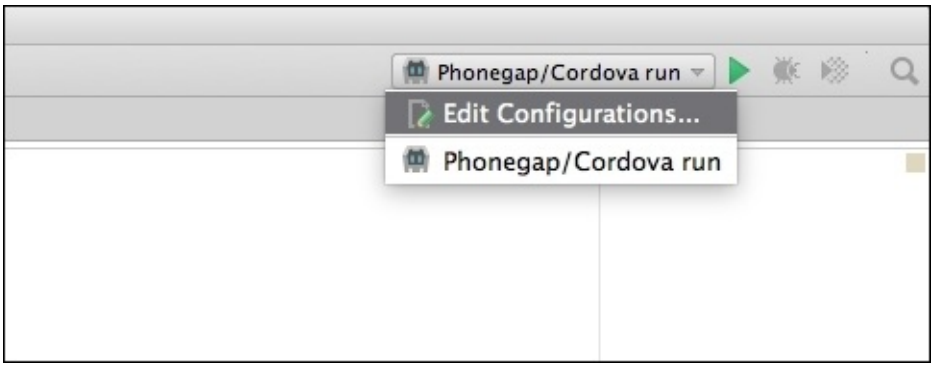
The preceding command installs Apache Cordova globally, so it will be available for all future projects. Once the installation is complete, we will have to restart WebStorm so that it detects the newly installed framework.

Now select **New Project** from either the welcome screen or the **File** menu. Select **PhoneGap/Cordova App** as a template, insert the destination folder, and select the path to the Cordova CLI, as shown in the following screenshot:



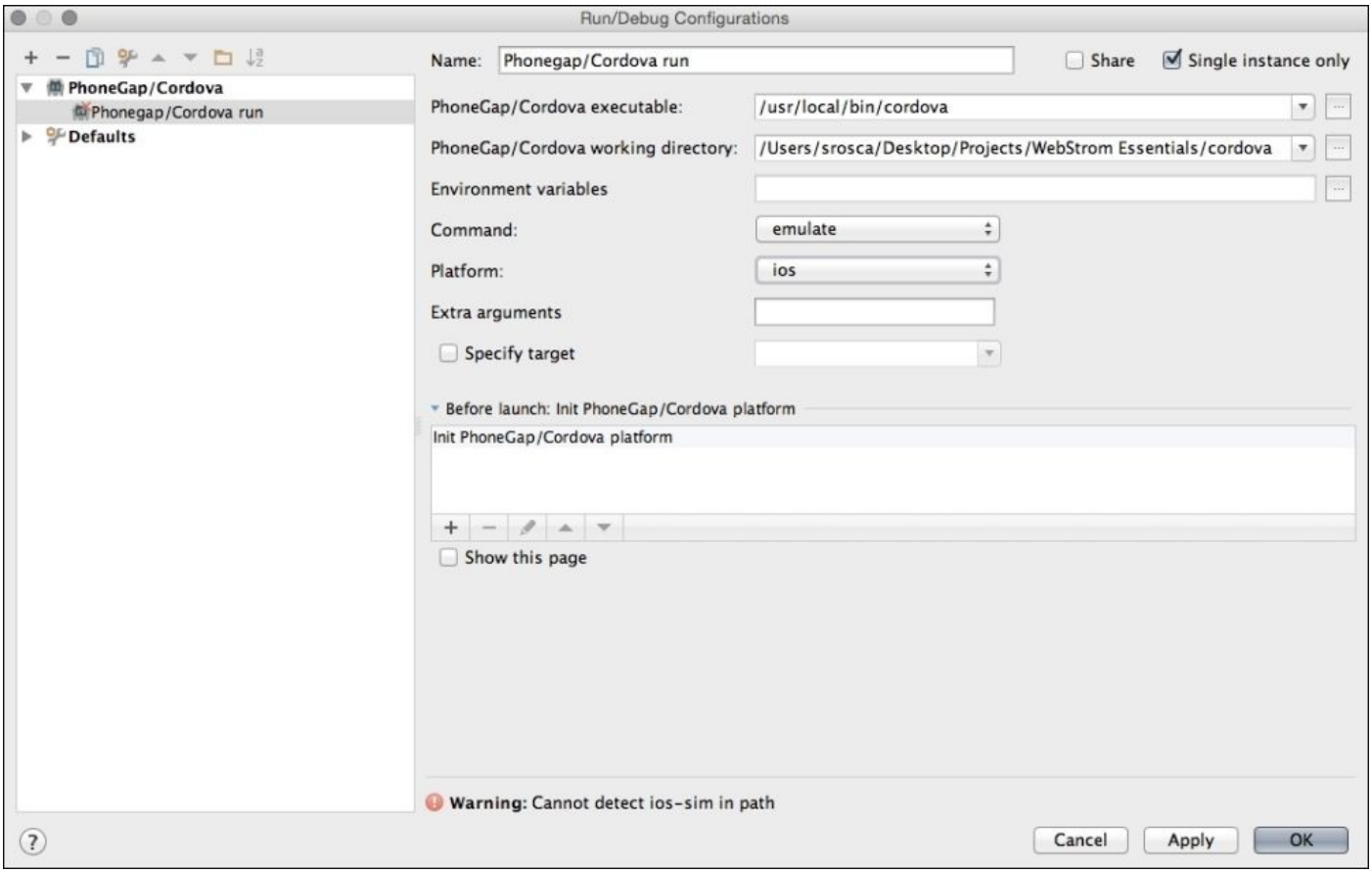
This creates and opens a new project based on the Cordova framework. The project comes preloaded with an example page that we can use.

Before testing our application in the emulators, we need to edit the run configurations. So select **Edit Configurations…**, as shown in the following screenshot:

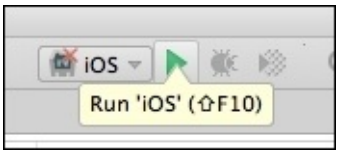


Change the name to iOS, and select **ios** in the Platform dropdown. Click on **OK** to save the configuration. This will create a run environment configuration that can be used to test the application on the iOS platform.

If we check **Specify target** (seen in the following screenshot), we can choose a specific device like iPhone 6, iPhone 5s, or others:



Now we can run our application in the emulator by pressing *Shift + F10* or selecting the run icon next to the configuration:

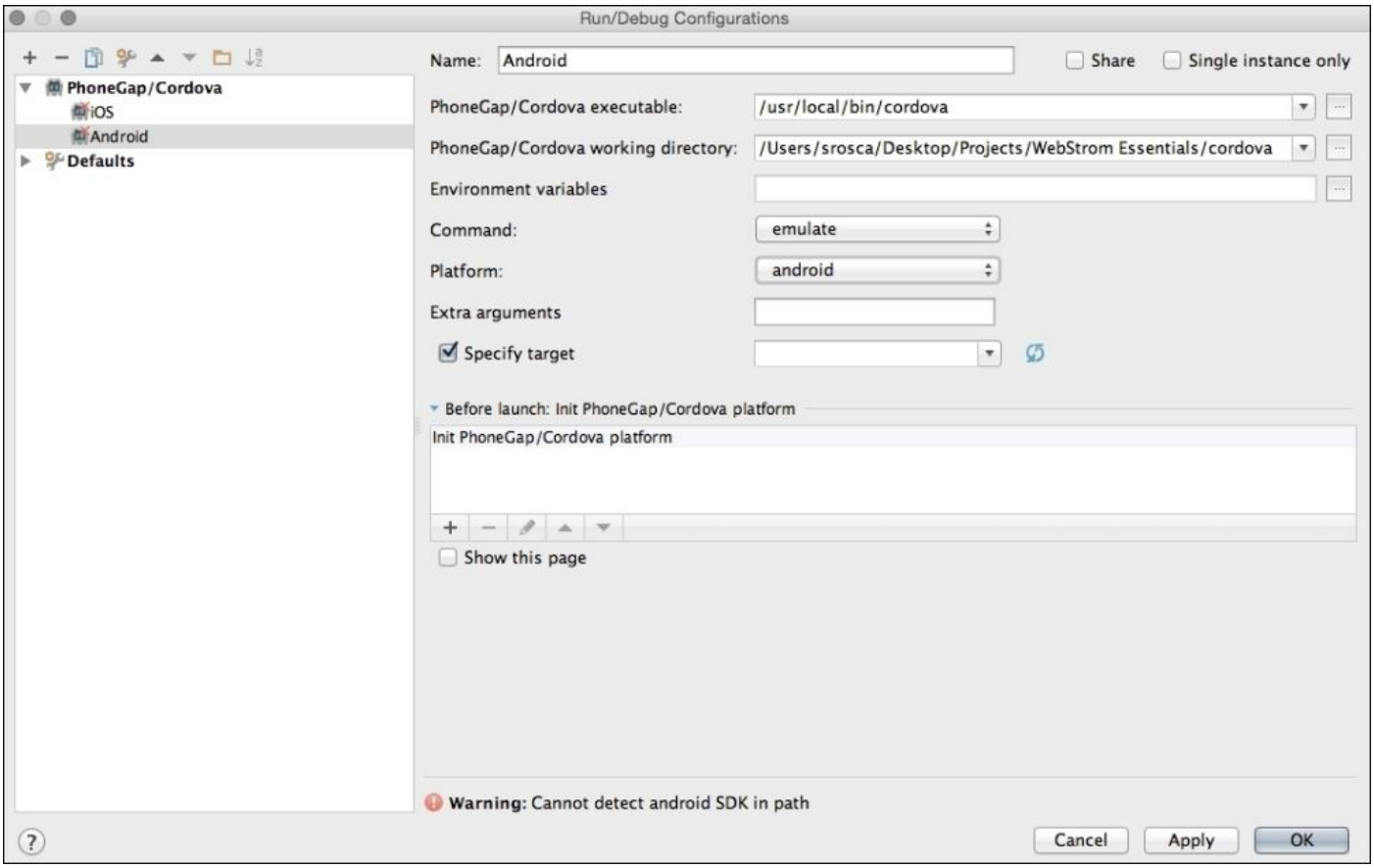


This will start the Cordova build system and run our application in the iOS emulator. We can also see the run log in the run section:

We are going to create a similar configuration to test our application on the Android platform. Select the **Edit Configuration** menu again.

With the previous iOS configuration selected, click on the **Copy Configuration** icon to duplicate it. In the newly-created configuration, change the name to `Android` and the

Platform to **android**.



We have now created a configuration to test our application on the Android emulator. To run the application, press *Shift + F10*, or press the run icon. This starts the application, using the Android emulator this time.

We have now created a simple app that waits for the device-ready event, and renders a simple screen changing the state when the device is ready. We are now going to build a more complex application using PhoneGap, since the differences between the frameworks are minimal.

PhoneGap

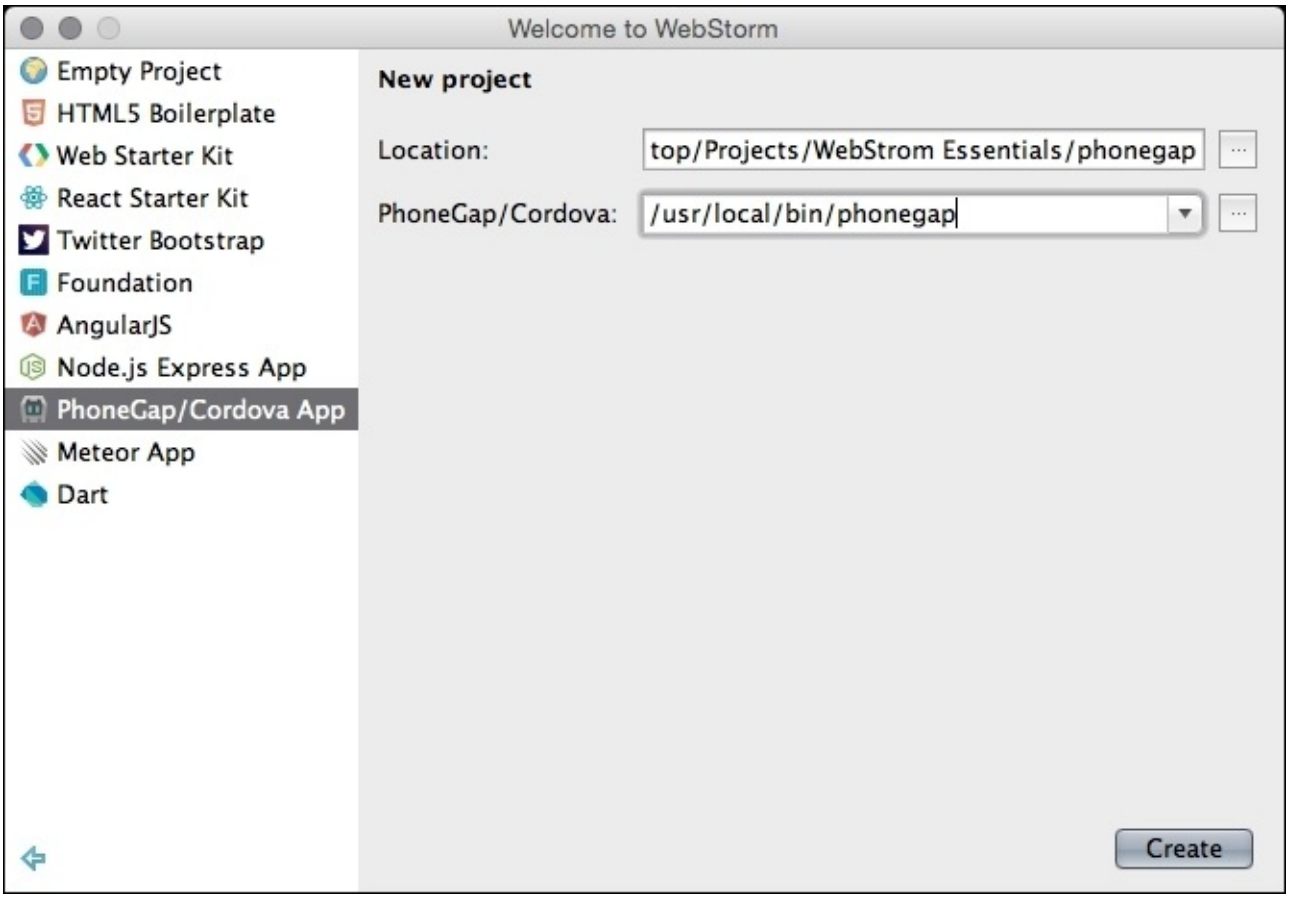
> *Easily create apps using the web technologies you know and love: HTML, CSS, and JavaScript PhoneGap is a free and open source framework that allows you to create mobile apps using standardized web APIs for the platforms you care about.*
>
> —http://phonegap.com/

PhoneGap is a distribution of Apache Cordova; so, the engine behind it is Cordova, but you have access to more tools such as the Developer App or remote builds.

Before we start, we need to install the framework globally, and restart WebStorm to pick it up. To do that, execute the following command:

```
npm install -g phonegap
```

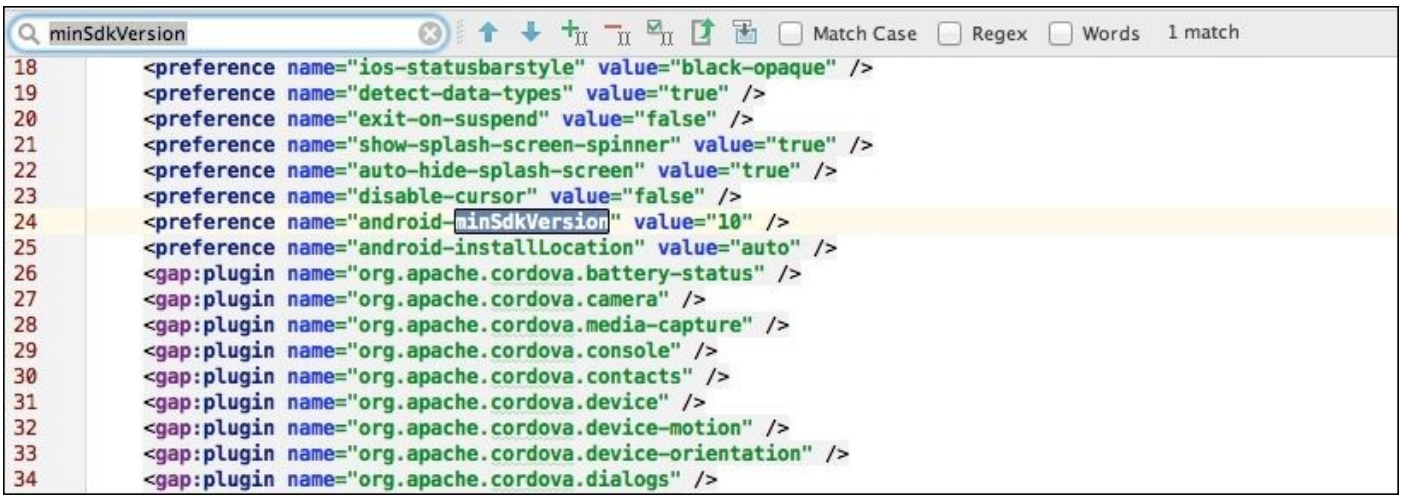
To start a new project based on PhoneGap, select **New Project** from either the welcome screen or the **File** menu, and select **PhoneGap/Cordova App** as the template. Insert the destination folder, and select the path to the newly installed PhoneGap CLI:



This creates and opens a new project based on the PhoneGap framework. This project comes with an example page that we are going to extend.

Before moving further, we need to change the **minSdkVersion** in the **config.xml** file to **10**. We do this, because, by default, Android Studio doesn't download the SDK for the old API versions, and we don't want to target the old version of Android.

To open the file, press ⌘ + *Shift* + *N*, start typing the file name (`config.xml`), and then select it. Once you have the file open, you can press ⌘ + *F*, and a search dialog will appear at the top of the page where we can type the `android-minSdkVersion` to find the fragment in the code.



Since our app uses the location, we need to add `cordova-plugin-geolocation` to the `config.xml` file so that PhoneGap downloads and installs it.

```
<plugin name="cordova-plugin-geolocation" version="1" />
```

Now, following the steps given in the previous section, we are going to create a run configuration for iOS and Android to test our application on both platforms.

In this section, we will create the map example from the first chapter, but this time as a mobile application.

Since our app uses jQuery, we need to download and copy it to the `JS` folder, and then load it in the `index.html` file. Load the following script between the `cordova.js` and `index.js` files:

```
<script type="text/javascript" src="js/jquery-1.11.3.min.js"></script>
```

After the app `<div>`, create a new `<div>` container for the map, as follows:

```
<div id="map"></div>
```

We also need to add the following styles to the `index.css` file:

```
html, body{
    width: 100%;
    height: 100%;
}

#map{
    display: none;
    width: 100%;
    height: 100%;
    margin: 0;
    padding: 0;
```

```
}
```

Now load the Google Maps API. Since the API is an external resource that will be downloaded, we will load it only once the device is ready and connected to the Internet. So open the `index.js` file, and add the following code inside the `onDeviceReady` handler:

```
onDeviceReady: function () {
    app.receivedEvent('deviceready');document.addEventListener("online",
app.loadGoogleMapsAPI, false);
    document.addEventListener("resume", app.loadGoogleMapsAPI, false);
    app.loadGoogleMapsAPI();
},
```

This will listen for when the device is online, and trigger the `loadGoogleMapsAPI` function that we are going to define next:

```
// Load the Maps API if it wasn't loaded before loadGoogleMapsAPI: function
() {
    if (window.google !== undefined && window.google.maps) {
        return;
    }
    // load maps api
    $.getScript('https://maps.googleapis.com/maps/api/js?
sensor=true&callback=app.onMapsApiLoaded');
},
```

The preceding function checks whether the API is loaded or not; if not, it loads it using jQuery's `getScript` method. Considering that the API was loaded asynchronously, we need to use the callback to pass as a query string parameter to the script.

Now we will create the `onMapsApiLoaded` callback function, where we get the device location. We are also going to watch for any changes in the location and read the location:

```
// Get the location after we have the maps api// Get the default location
and also watch for any changes in the location
onMapsApiLoaded: function(){
    navigator.geolocation.getCurrentPosition(app.geolocationSuccess,
app.geolocationError,
app.geolocationOptions);navigator.geolocation.watchPosition(app.geolocation
Success, app.geolocationError, app.geolocationOptions);
},
```

For the location method, we have to create two callback functions and a configuration object. We will use the default location if there is an error while loading the location:

```
// Load the map with the device positongeolocationSuccess:
function(position){
    app.position = new google.maps.LatLng(position.coords.latitude,
position.coords.longitude);
    app.updatePosition();

},
// If we don't have a position from the device, we are going to use // some
default values
geolocationError: function(error){
    if(!app.pos){app.defaultPosition = new google.maps.LatLng(51.5, -0.1);
```

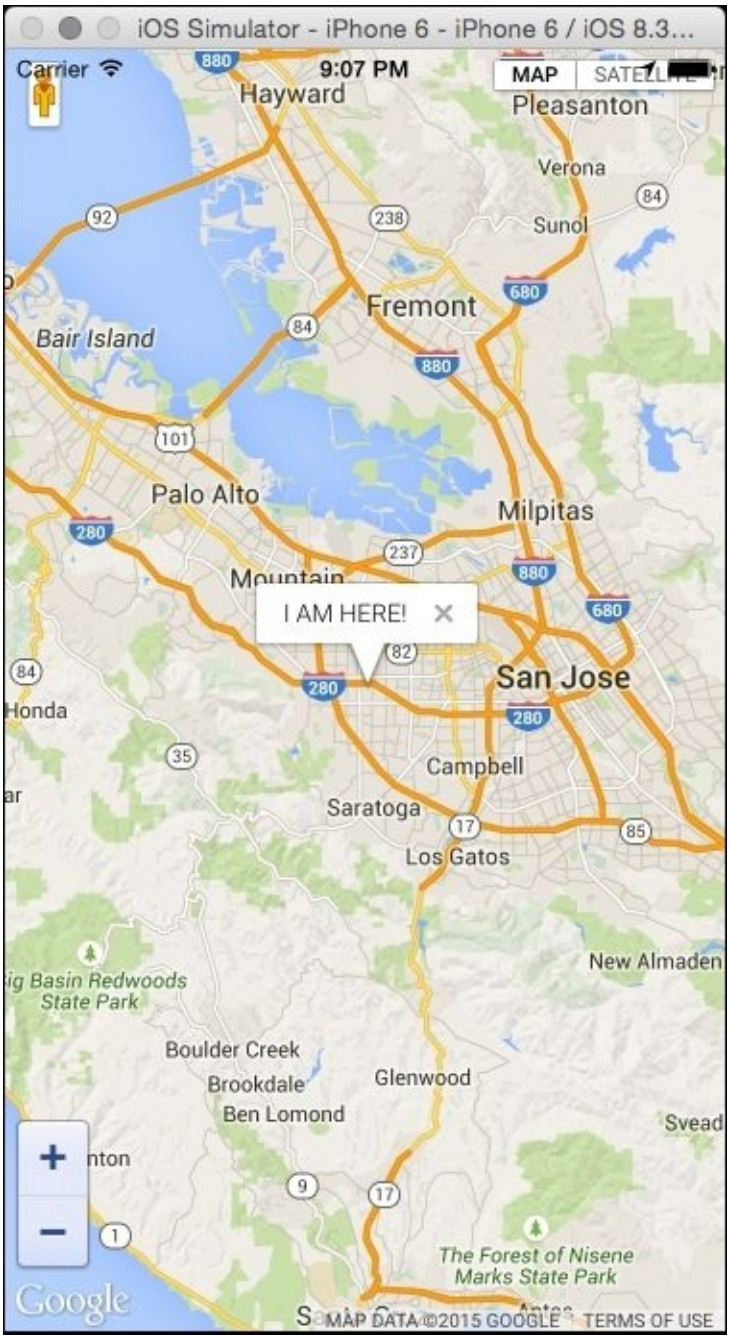
```
        app.updatePosition();
    }
},
// Settings object for the geolocationgeolocationOptions: {
    maximumAge: 3000,
    timeout: 5000,
    enableHighAccuracy: true
},
```

Once we have the location from the device or the default one, we can create and update the map.

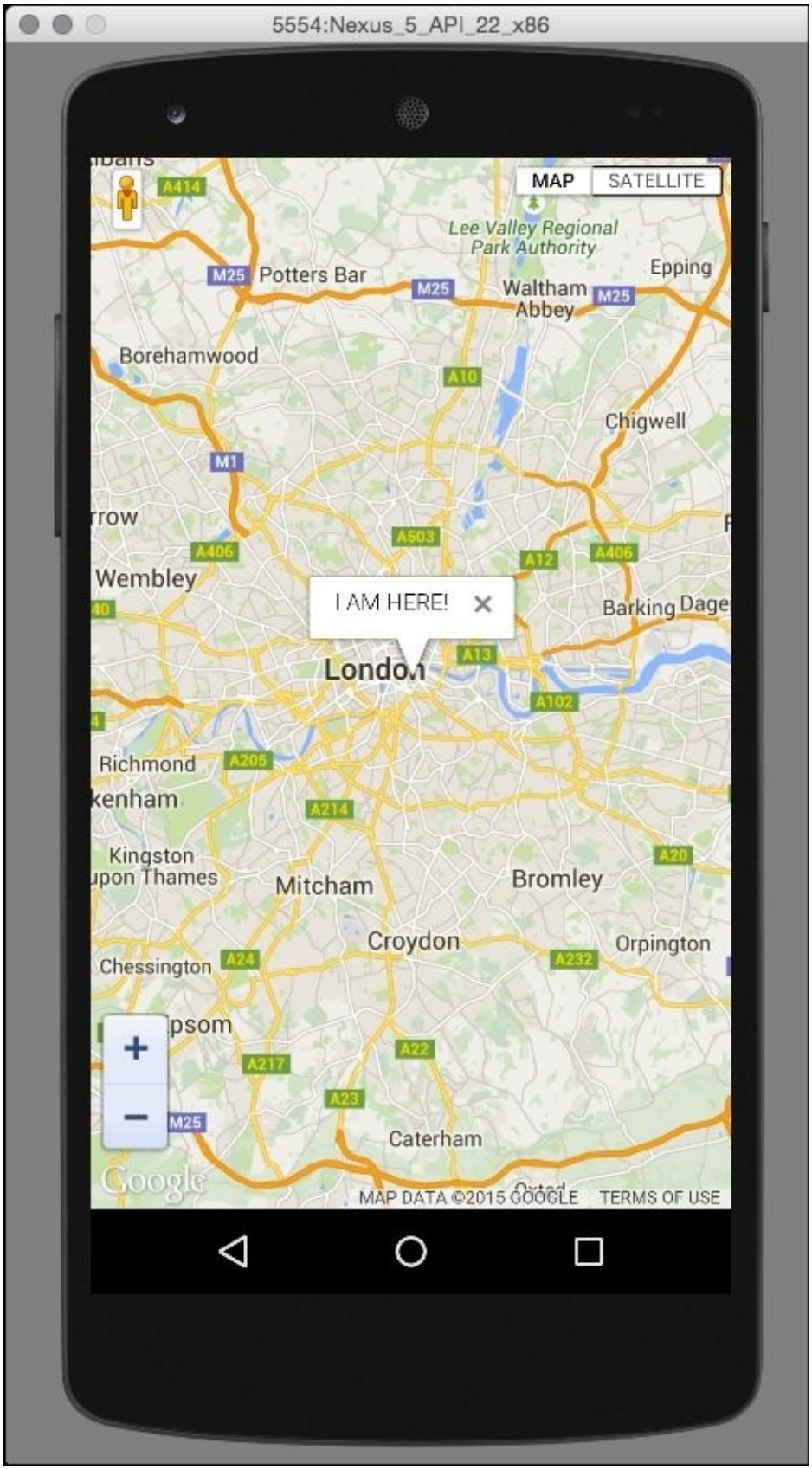
```
// Show the map and update the positionupdatePosition: function(pos){
    $('.app').hide();
    $('#map').show();
    var mapOptions = {
        zoom: 10
    };
    var pos = app.position || app.defaultPosition;
    var map = new google.maps.Map(document.getElementById('map'),
mapOptions);
    var infowindow = new google.maps.InfoWindow({
        map: map,
        position: pos,
        content: 'I am here!'
    });
    map.setCenter(pos);
},
```

This will hide the initial screen, create the map, and update our position on it.

Now, if we follow the steps from the previous example for running the application in the iOS emulator, we will see the following screen:



If we run the application in the Android emulator, we will get a similar screen:


5554:Nexus_5_API_22_x86
MAP
SATELLITE
Lee Valley Regional Park Authority
Potters Bar
Waltham Abbey
Epping
Borehamwood
Chigwell
Wembley
I AM HERE!
Barking
London
Richmond
Kingston upon Thames
Mitcham
Bromley
Croydon
Orpington
Chessington
Caterham
MAP DATA ©2015 GOOGLE
TERMS OF USE

The Ionic framework

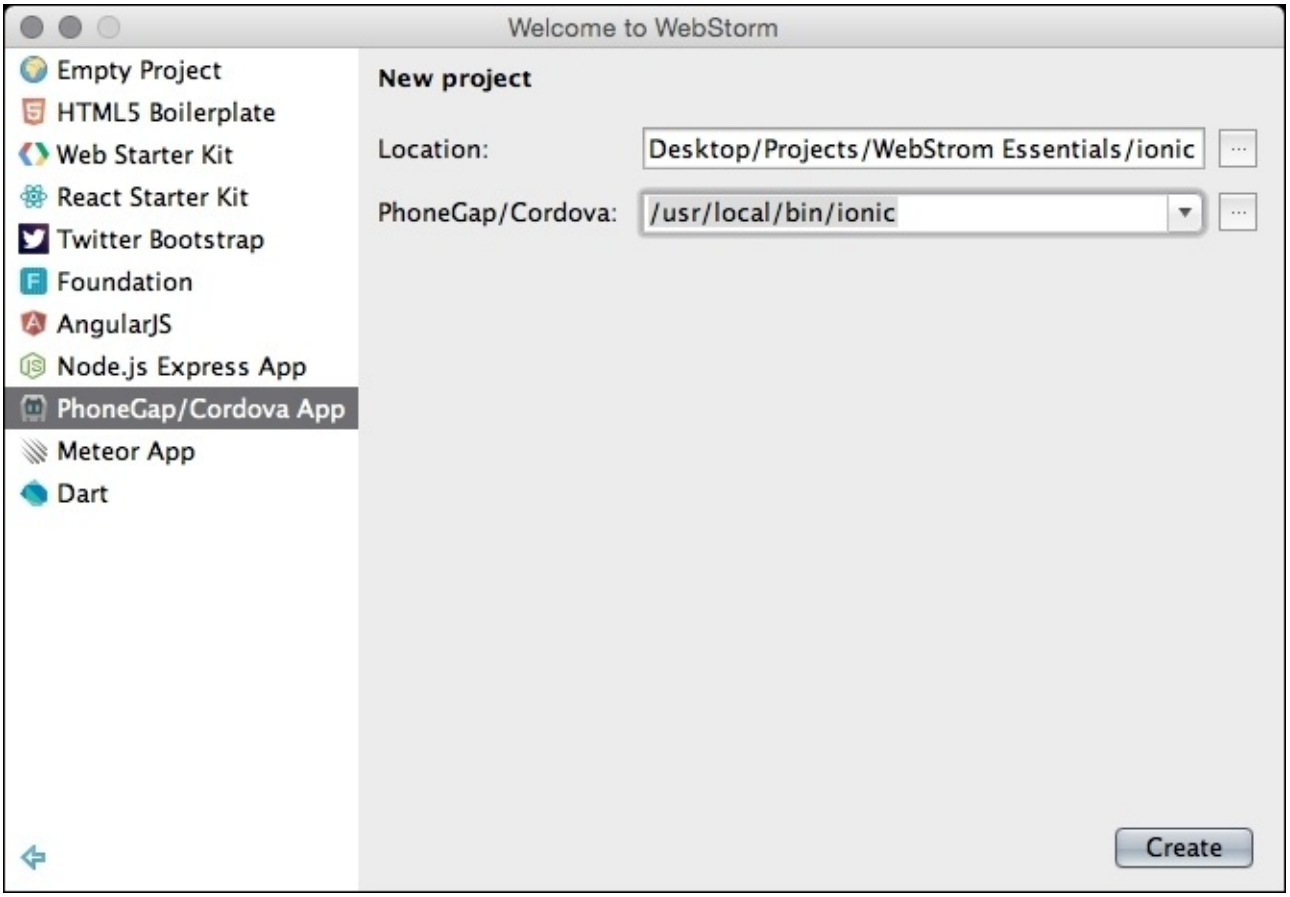
> *Create incredible apps.*
>
> *Iconic is the beautiful, open source front-end SDK for developing hybrid mobile apps with web technologies.*
>
> —http://ionicframework.com/

Ionic is a framework that focuses on the look, feel, and UI interactions. It is not a replacement for PhoneGap or Cordova, but an addition to these. It can be viewed as the Bootstrap for mobile development, since it is a collection of CSS styles and Angular JavaScript components.

Before we start, we need to install the framework globally and restart WebStorm to pick it up. To do that, run the following command:

```
npm install -g ionic
```

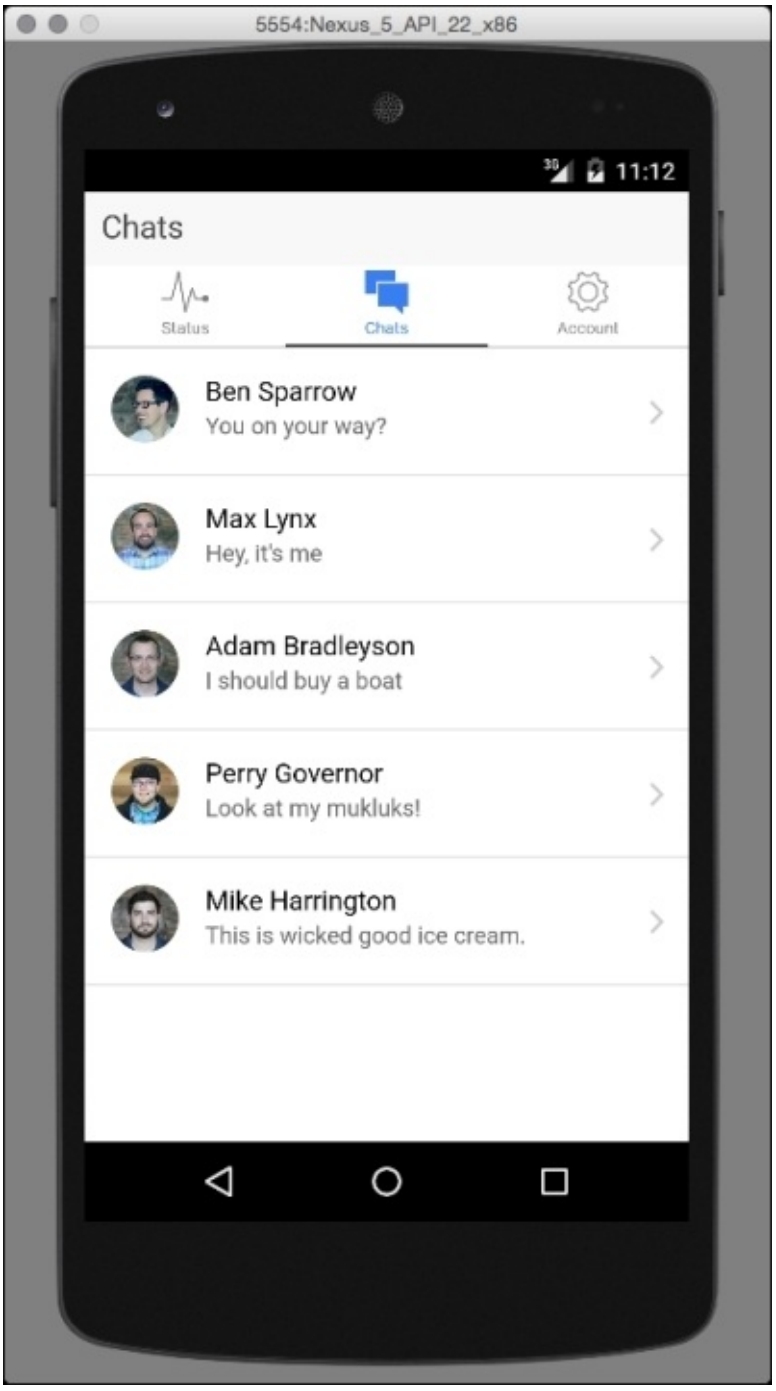
Once we have installed the application, we need to restart WebStorm so that it can pick up the framework, and start a new project selecting the **PhoneGap/Cordova App** template and using the newly-installed Ionic CLI as the project compiler:



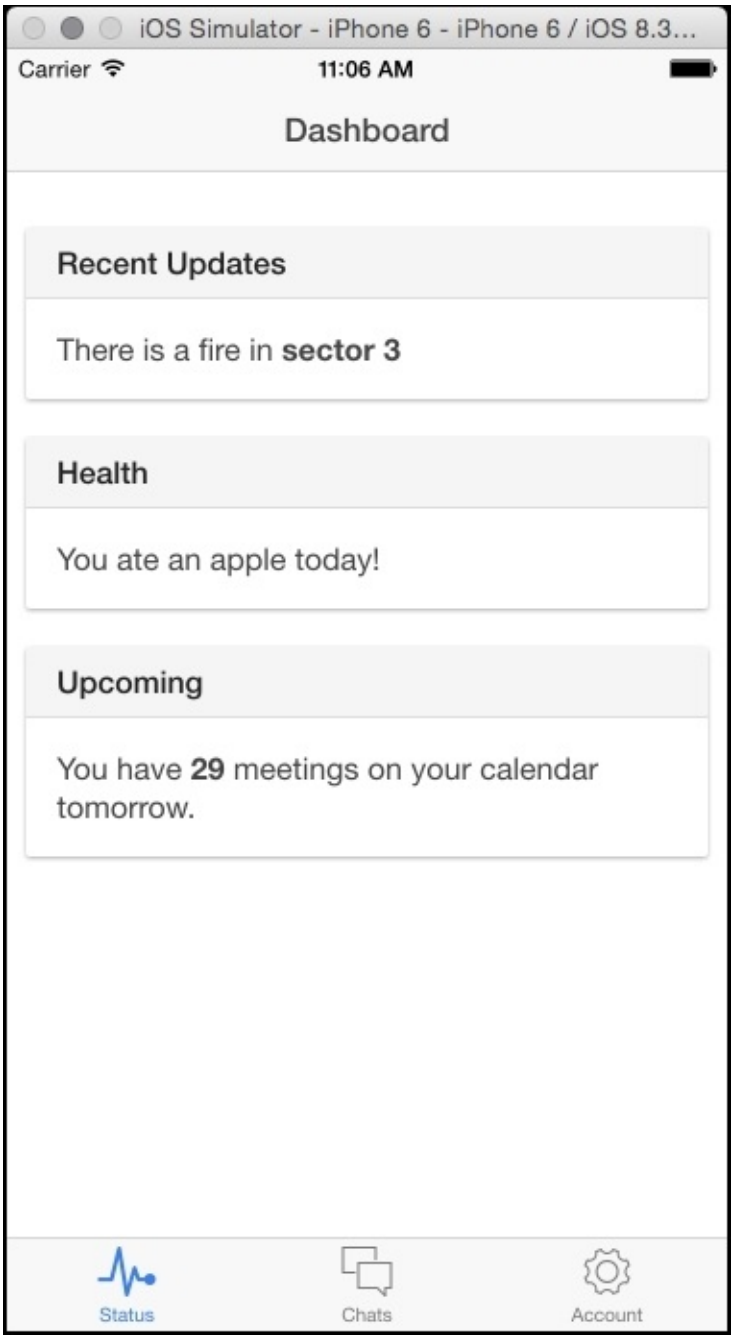
After the project is created, we need to create the run configuration for both iOS and Android in the same way that we did for the previous projects; so, create the configuration following the steps from the previous section.

Since Ionic comes with a sample project, we can readily test the created project on the

emulator.



This framework will create a more complex application with a built-in navigation that will allow us to see more screens.


iOS Simulator - iPhone 6 - iPhone 6 / iOS 8.3...
Carrier
11:06 AM
Dashboard
Recent Updates
There is a fire in sector 3
Health
You ate an apple today!
Upcoming
You have 29 meetings on your calendar tomorrow.
Status
Chats
Account

Summary

In this chapter, you have learned how to use web technologies to create native mobile applications, and how WebStorm assists us in doing so: from creating projects quickly using templates to managing the run configurations.

In the next chapter, we are going to see how WebStorm can help us in analyzing and debugging our code.

Chapter 7. Analyzing and Debugging Your Code

In the previous chapter, we learned how to use web technologies to create native mobile applications, and how to create projects quickly with the help of templates. In this chapter, we are going to see how WebStorm can help us analyze and debug our code.

We will now focus on code quality tools like Code Inspector, Code Style, and Code Linters that will help us write quality code. We will also learn to debug our code directly inside the IDE. In this chapter, we will cover the following topics:

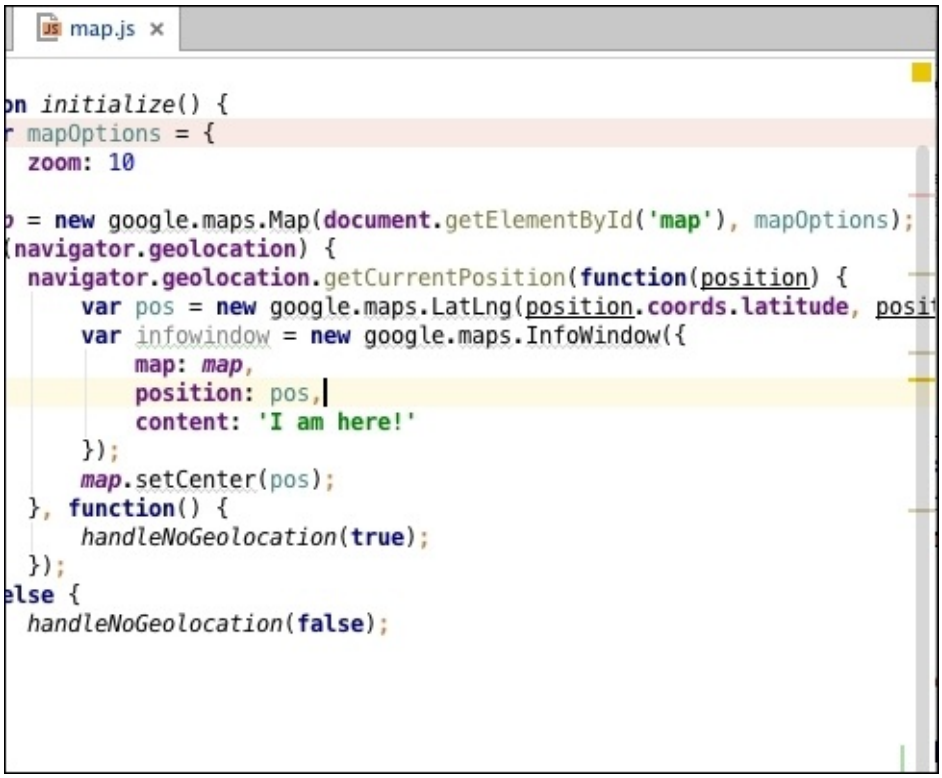
- Understanding code inspection
- What is Code Style?
- Using different code quality tools
- Debugging your code

Code inspection

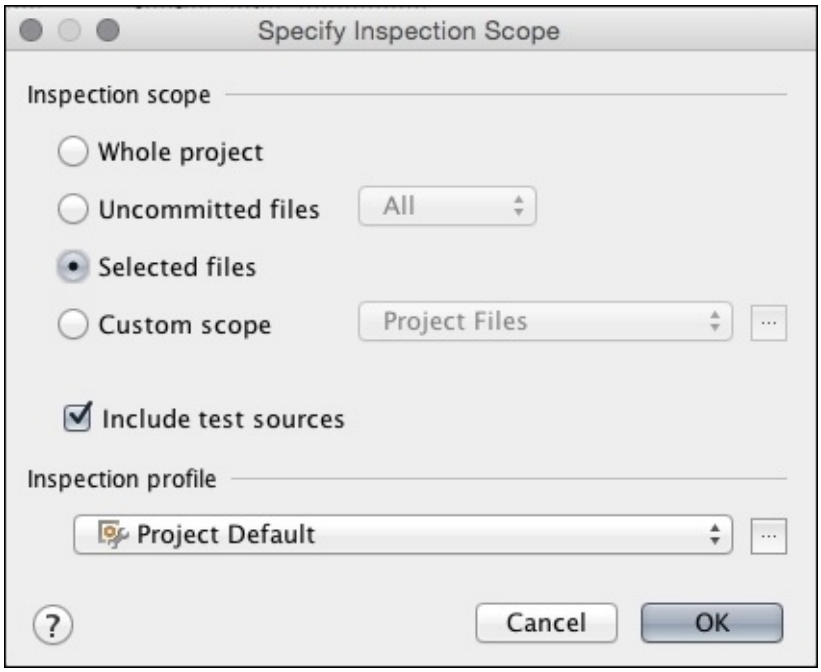
WebStorm is equipped with a powerful, fast, and versatile code analysis tool. This tool detects not only the compiler and runtime errors, but also the different code inefficiencies. It suggests corrections and enhancements while you code. Some of the common errors it can detect are: unreachable code, unused code, non-localized strings, unresolved methods, memory leaks, and spelling problems.

The inspection is performed in multiple ways:

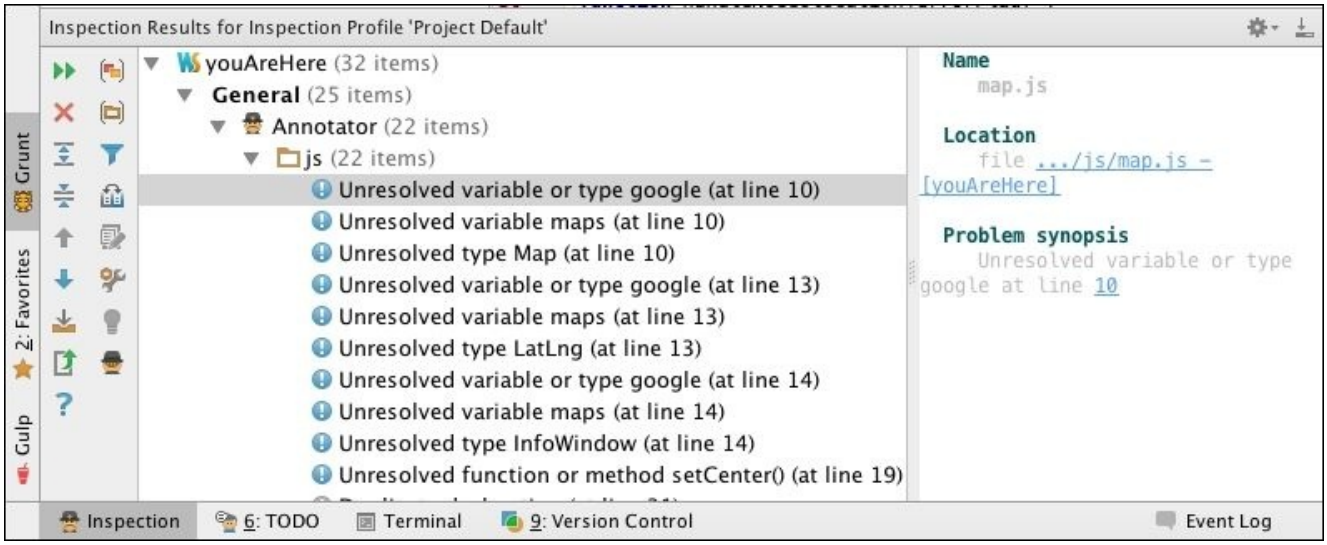
- Analyzing all opened files, all the code issues are highlighted directly in the editor. The status is visible on the right side of the editor. At the top of the file and on the scroll bar is a summary that takes you to the line of code when clicked.



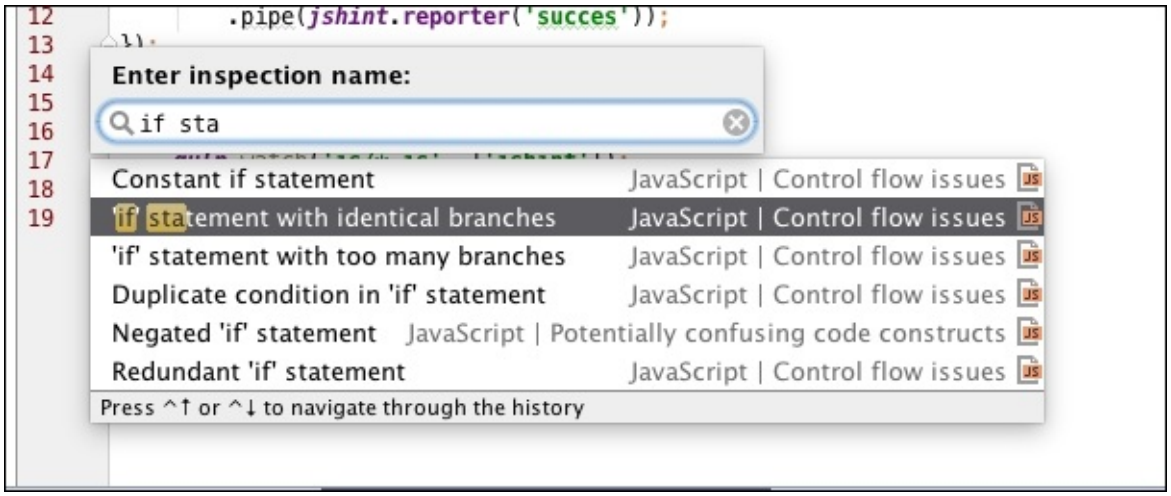
- Running the analysis in bulk mode runs the inspector in a specified scope. The scope can be any set of files, from the currently selected ones to the whole project. To run the bulk mode, select **Code** | **Inspect Code…**, and specify the scope in the opened dialog.



In the Specify Inspection dialog, we can also specify the **Inspection profile** that we want to use. The profiles represent a set of rules that we want our code to be checked against. For now, we are going to leave it to the default one, and click on **OK**. WebStorm will run the inspection, and then present us with a summary dialog, as seen in the following screenshot:

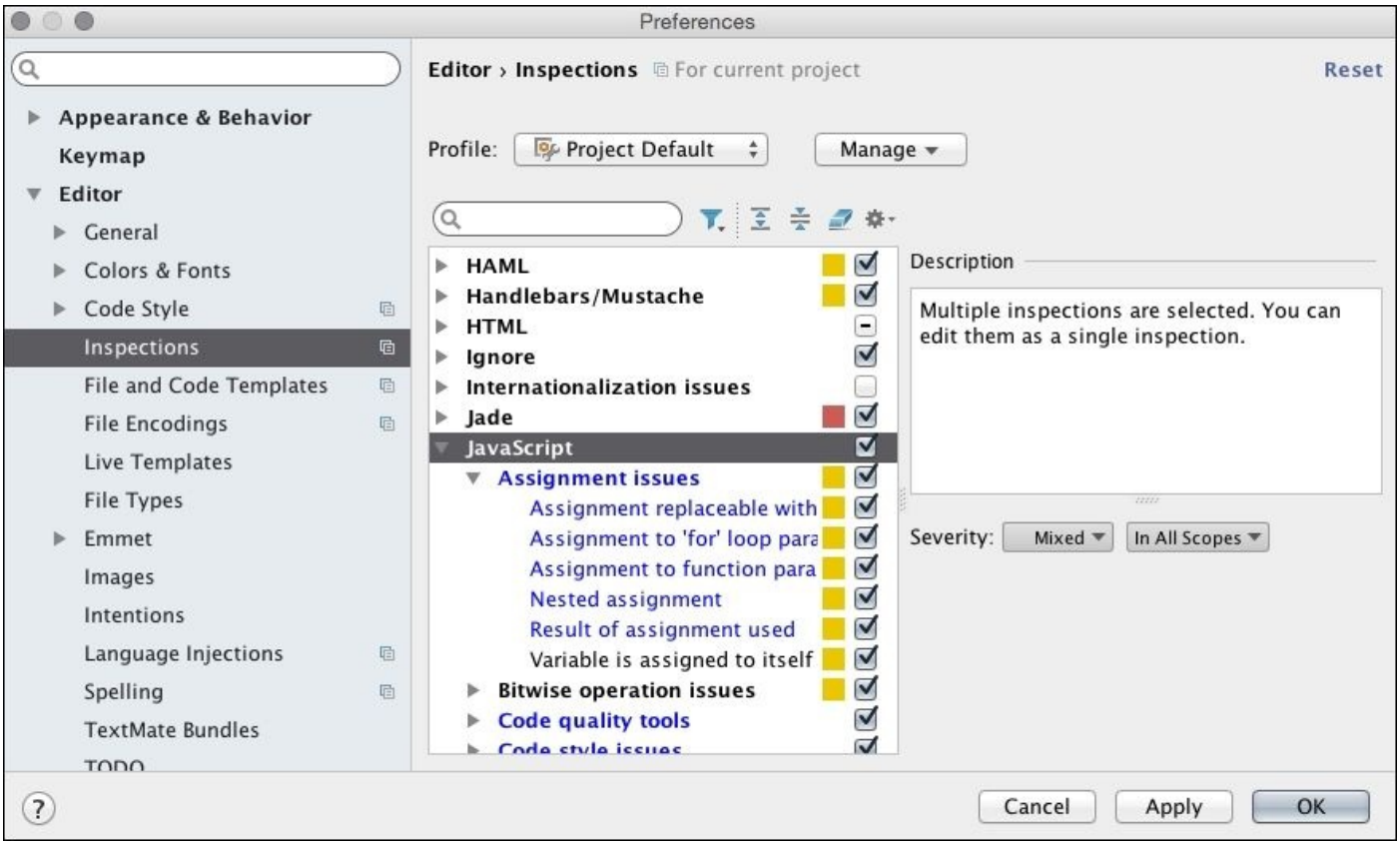


- Running a single code inspection in a specified scope—in this mode, WebStorm allows us to check for a specific problem in the selected scope. To access the dialog, open **Code | Run Inspection By Name**, and select the inspection that you want to run in this dialog, as seen in the following screenshot:



This opens the scope dialog, where we can select the code on which this inspection is to be applied. WebStorm then takes us to the summary page, similar to the bulk section.

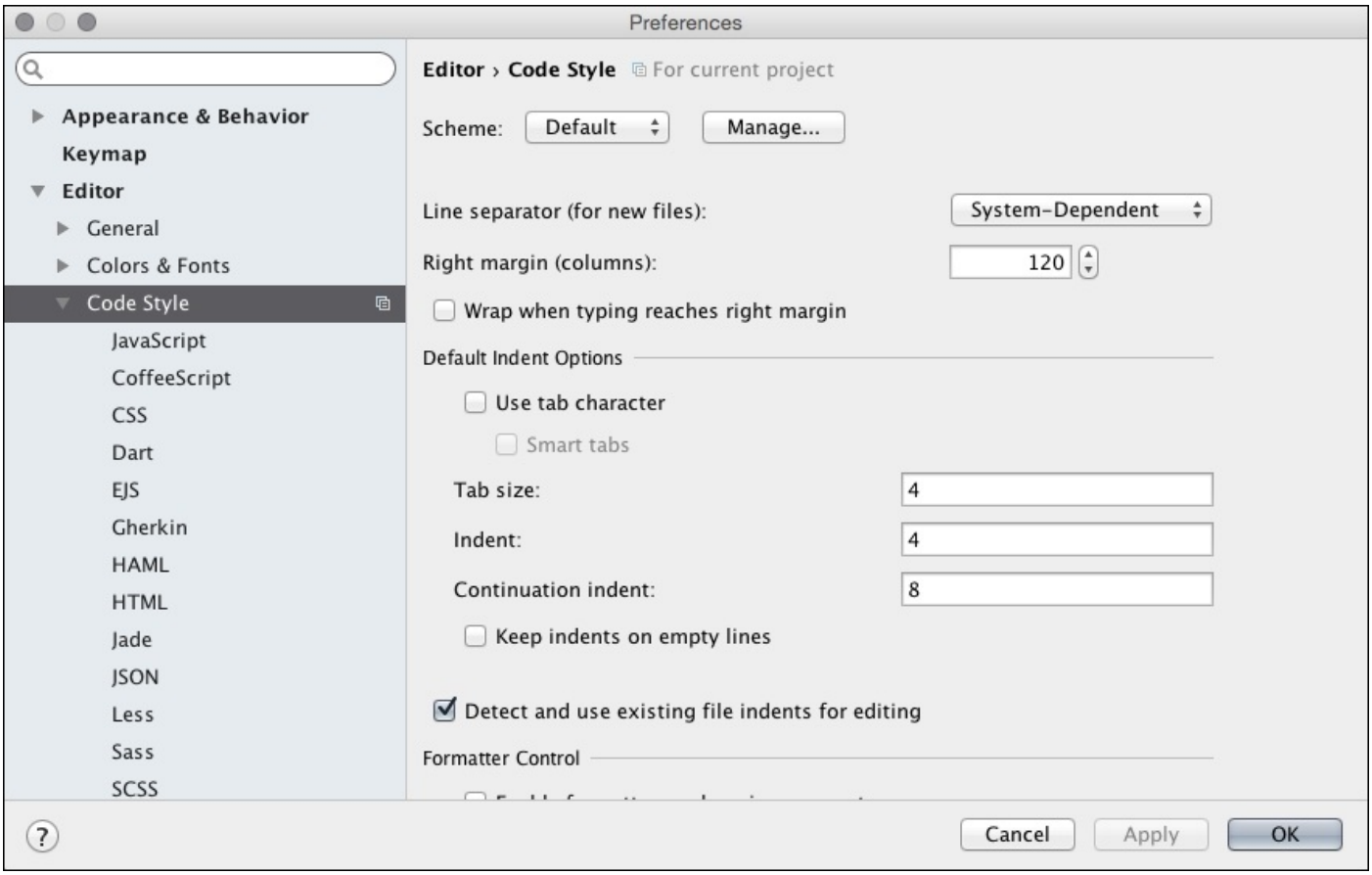
WebStorm works with profiles when inspecting the code, so you can specify the type of problems that you want to search and report. It also allows you to define profiles at the application or project level, and to share the created profiles with the rest of the team. All the profile settings can be found in the **Preferences** | **Editor** | **Inspections** page.



In this section, we focused on the way your code performs and executes; in the next section, we are going to focus on the looks of code.

Code Style

The **Code Style** can be seen as a set of specific rules that dictate the layout of the source code. It includes the indentation, use of white space, the style used for variable names, keywords, and so on. WebStorm allows you to manage the scheme to be applied to specific files type. To access the dialog, go to **Preferences** | **Editor** | **Code Style**.



The first page that is displayed is for setting the general styles, like the following, for the current project:

- **Line separator** for columns and wrapping
- **Default Indent Options**
- **Formatter Control** for using comments to turn the Code Style on/off
- **EditorConfig** for controlling the styles through a general `.editorconfig` file that can be shared with the project

We can also set language-specific code styles to make the editor fit our needs. To change these settings, we need to select the desired language from the menu in the previews screen.

WebStorm also allows you to quickly apply the code style to an open file by selecting **Code** | **Reformat code**. It applies the necessary rules based on the file extension.

These tools help in giving a consistent look to your code, and with the help of the **EditorConfig** files, you can share your styles across multiple members of your team. It is

considered a good practice to include these files in the Git repository of your project.

Code quality tools

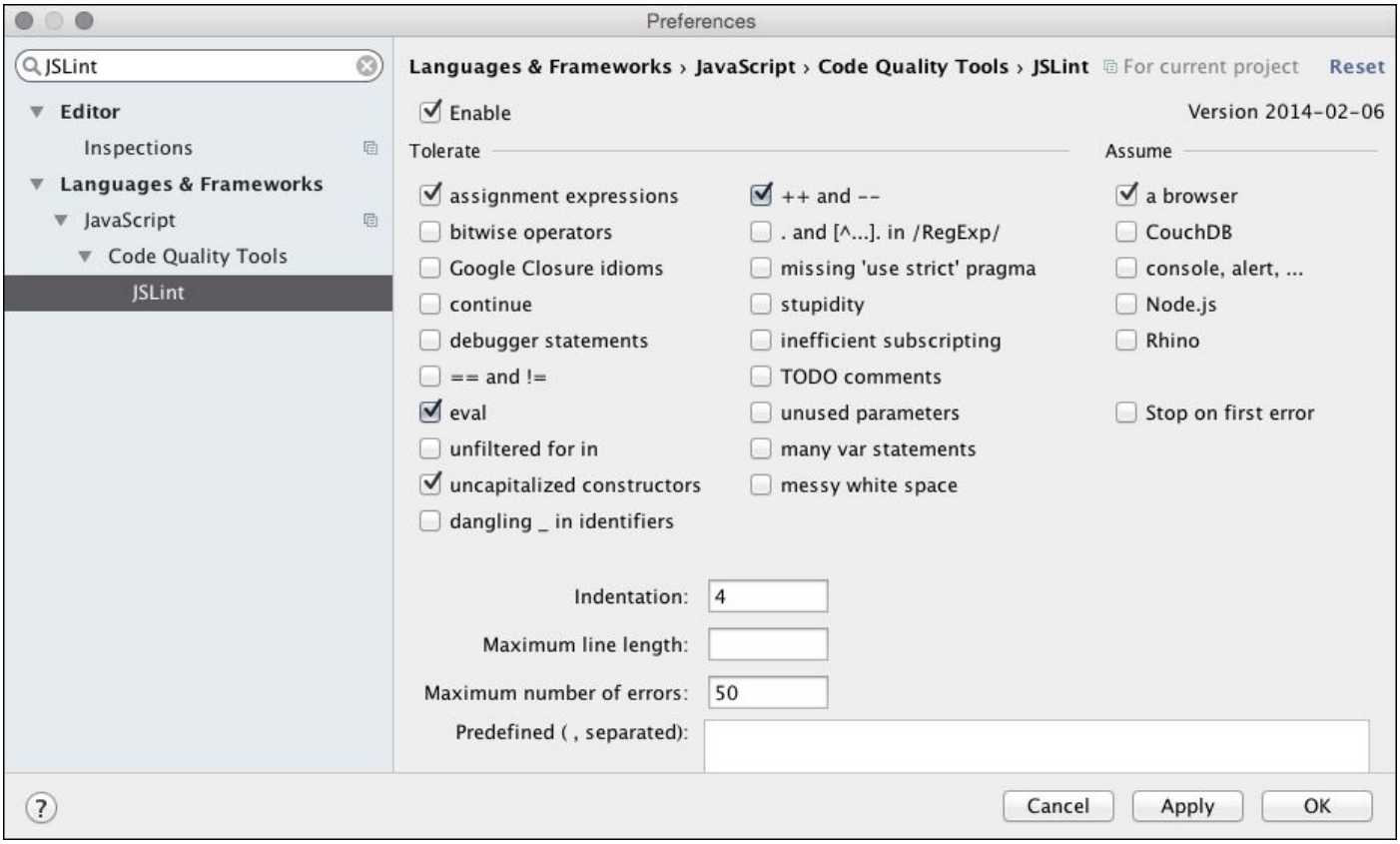
WebStorm allows us to use various code quality tools to make sure that our code is free from common mistakes and discrepancies. WebStorm 10 supports integration with the following:

- JSLint
- JSHint
- Closure Linter
- JSCS
- ESLint

It comes bundled with JSLint and JSHint, but the rest need to be installed through NPM or other installers. We are going to focus on JSCS and the first two tools in the preceding list, since these are the most common ones, and the workflow is very similar between the different tools.

JSLint

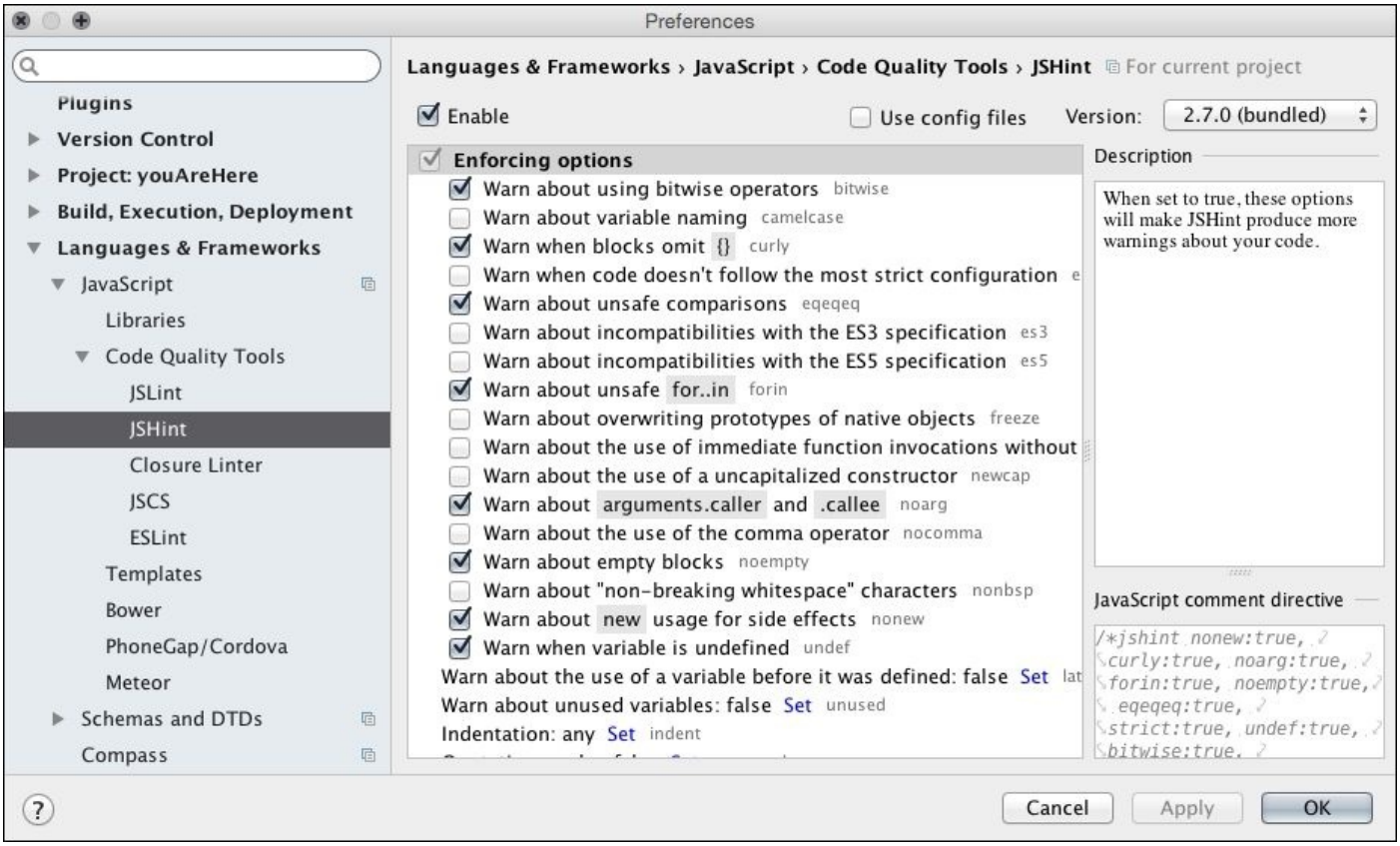
To use the JSLint tool, enable it in the **Settings** | **Languages & Frameworks** | **JavaScript** | **Code Quality Tools** | **JSLint** page, and select the desired settings. A good practice, when navigating to the complex settings page, is to use the built-in search.



Once we enable and configure JSLint, it automatically checks all the open JS files with the specified settings, and the errors are displayed like the code inspection messages.

JSHint

JSHint is another code quality tool that comes bundled with WebStorm. The settings page can be found at **Settings** | **Languages & Frameworks** | **JavaScript** | **Code Quality Tools** | **JSHint**.

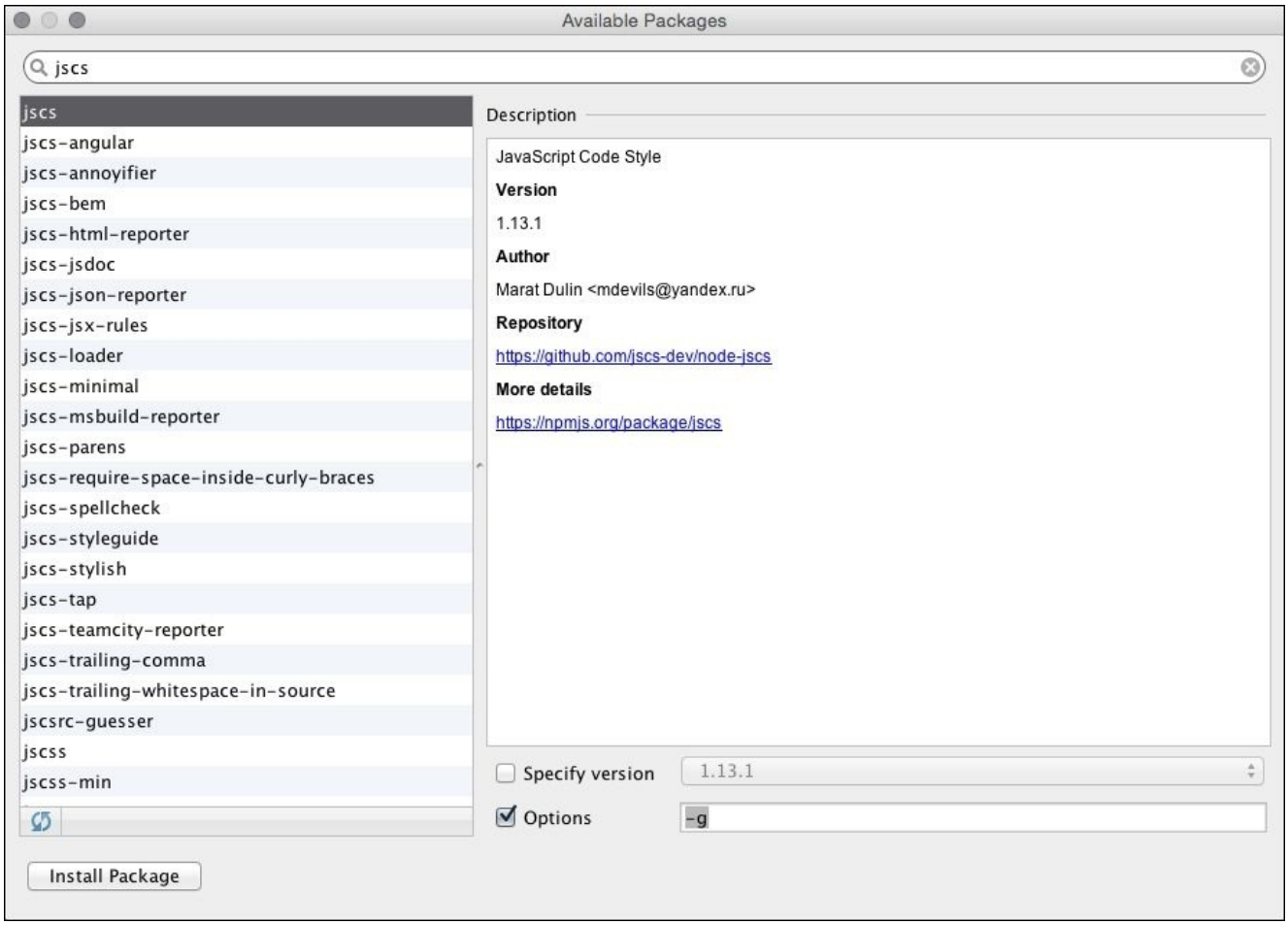


The JSHint settings can be configured either through the settings page, or by using a special `.jshintrc` configuration file. Again, it is a good practice to include such a file in your repository.

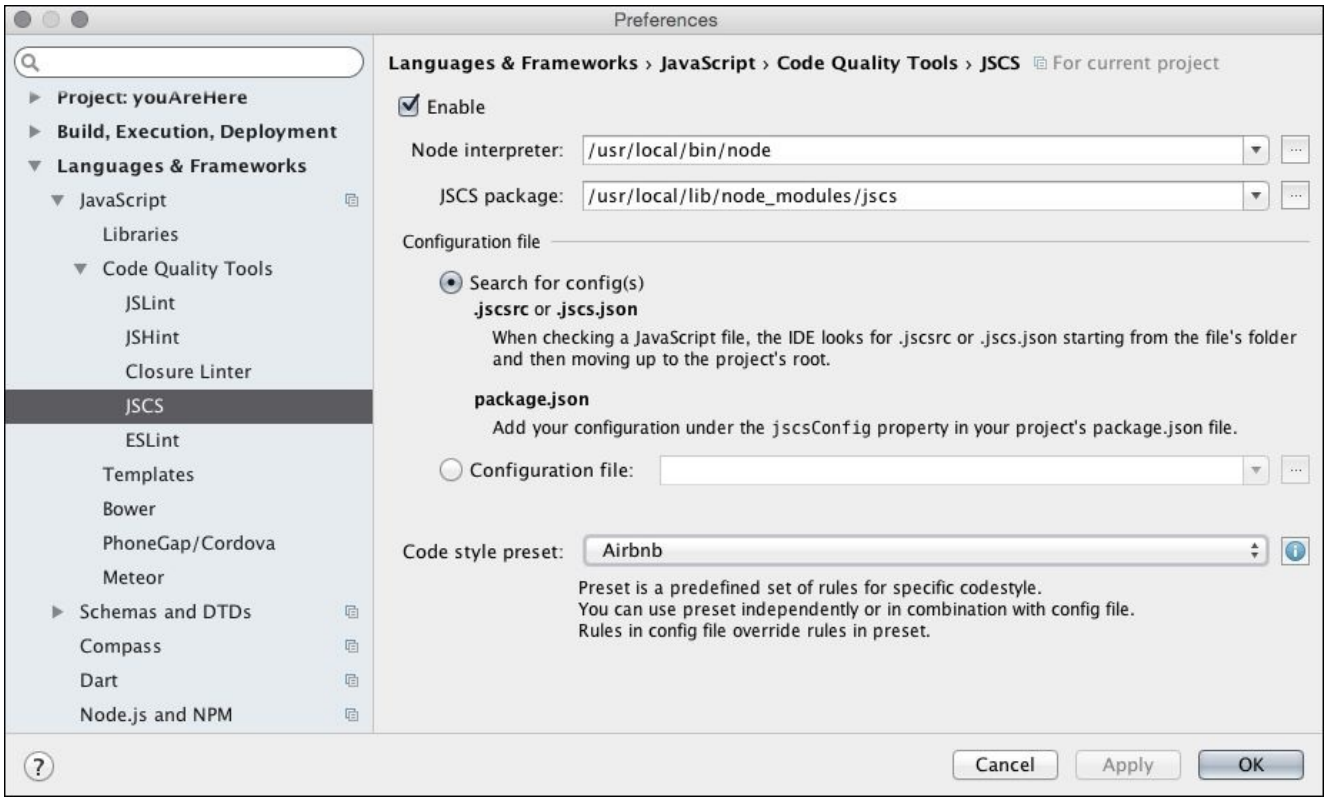
JSCS

JSCS is one of the popular code linters used by some of the big frameworks like jQuery, Bootstrap, and Angular. It doesn't come bundled with WebStorm like the previous tools, so the first thing we have to do is to install it. We are going to use the build in NPM installer, and perform the following steps:

1. Open the **Preferences** | **Languages & Frameworks** | **Node.js and NPM page** | **Packages Install** dialog, search for `jscs`, and install it globally.



2. Once the installation is ready, the settings page can be found at **Settings** | **Languages & Frameworks** | **JavaScript** | **Code Quality Tools** | **JSCS**.
3. On this page, select the package from the global node folder and the configuration file used for the rules or a preset. There is no option to define the rules individually like the previous tools.



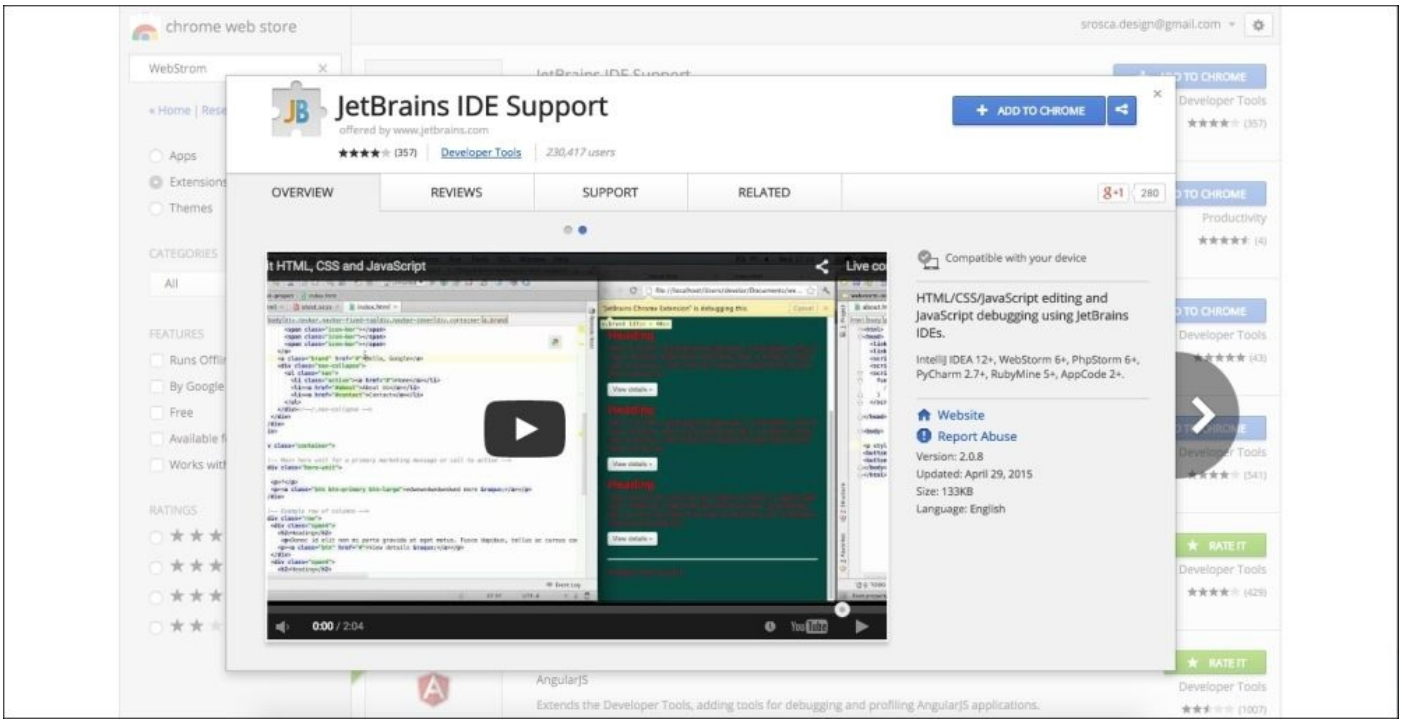
For this demonstration, we are going to use one of the popular set of rules, the one defined by **Airbnb**.

Now that you have learned how to check the code before it even runs, we are going to dive into the world of debugging and see how WebStorm can help us understand and fix our code while it is running.

Debugging your code

One of the first things that we need to do before starting is to install and configure the browser extension for Chrome. This extension enables WebStorm to communicate with the browser, so we can debug our application directly inside the IDE.

The Chrome browser extension is available at the **chrome web store** as **JetBrains IDE Support**. So, we need to add it from there, as shown in the following screenshot:



Now that we have the extension installed, we can start learning to debug our code. To be able to debug the code while it is running, we need to start a debugging session. There are several ways to start a debug session. In this section, we are going to see some of the ways of debugging in WebStorm.

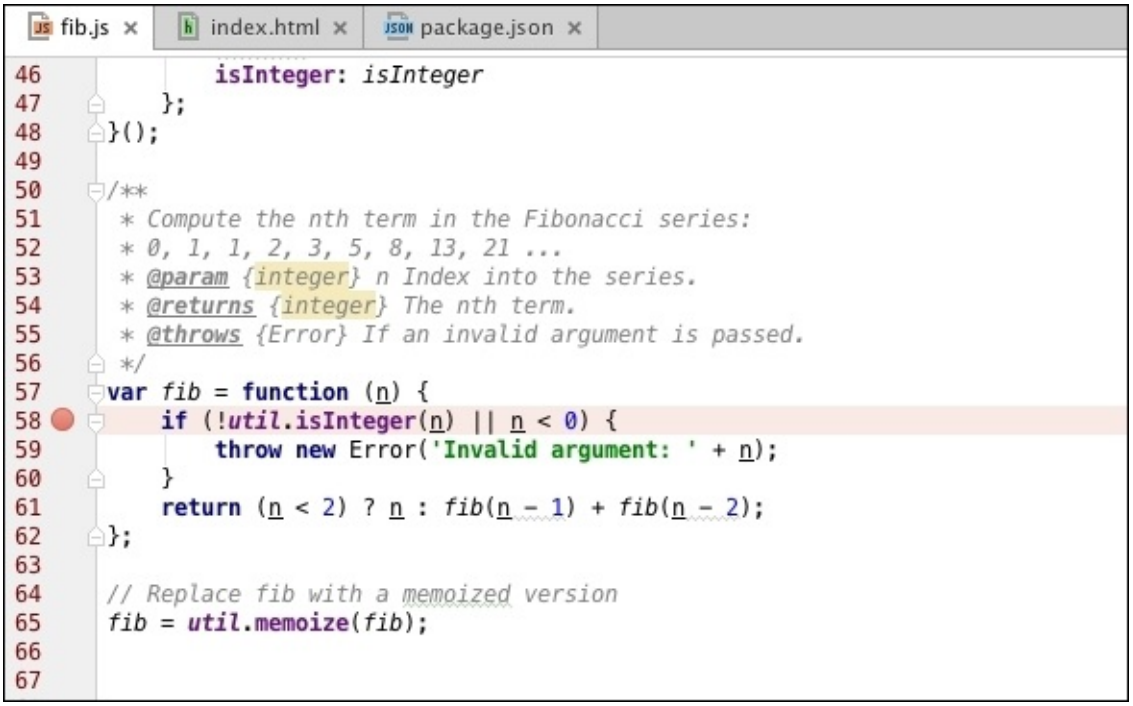
Initializing a debug session from the browser

First we need to open the project inside WebStorm, and set some breakpoints. Before we start, we need do import a a project created for this purpose from GitHub. So, using the instructions given in the previous section, create a project from the Git repository at https://github.com/srosca/webstorm_essentials_debugging.git. After you have downloaded and opened the project, run `npm install` to install all the required dependencies.

Note

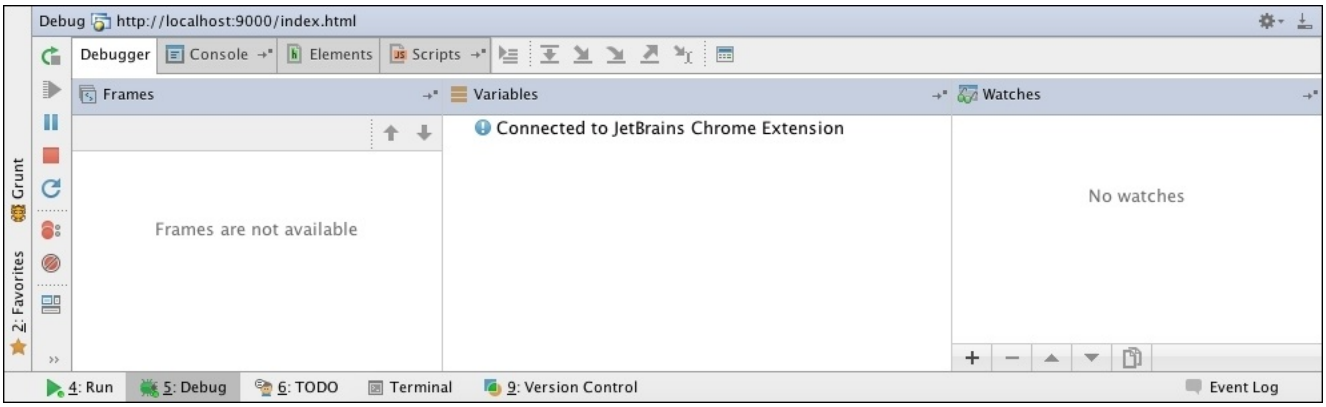
Once everything is installed, make sure that you start the sever by using the Grunt task runner or typing `grunt serve` inside the terminal.

Open the `fib.js` file and you can click on the right-hand side to set a breakpoint; this will look like a red dot. We are going to set a breakpoint on line 58 inside the `fib` function to check what happens inside.



Execute the following steps:

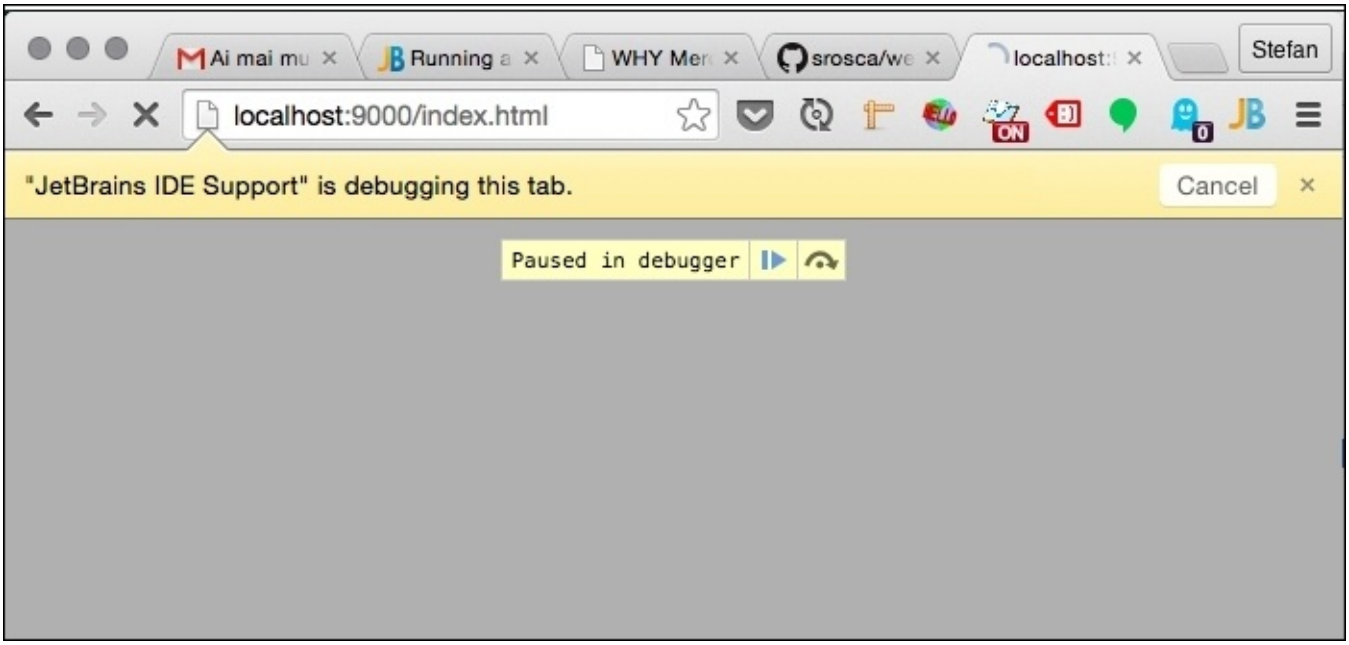
1. Open the Chrome browser and point it to `http://localhost:9000/index.html` to open the file.
2. In the browser, select **inspect** in WebStorm from the **context** menu. This creates a temporal debugging session, and links Chrome with WebStorm.
3. You will now see a message **JetBrains IDE Support is debugging this tab**, in the browser, and the debugging section with the message, **Connected to JetBrains Chrome Extension** in WebStorm.



The debug section is split into four tabs:

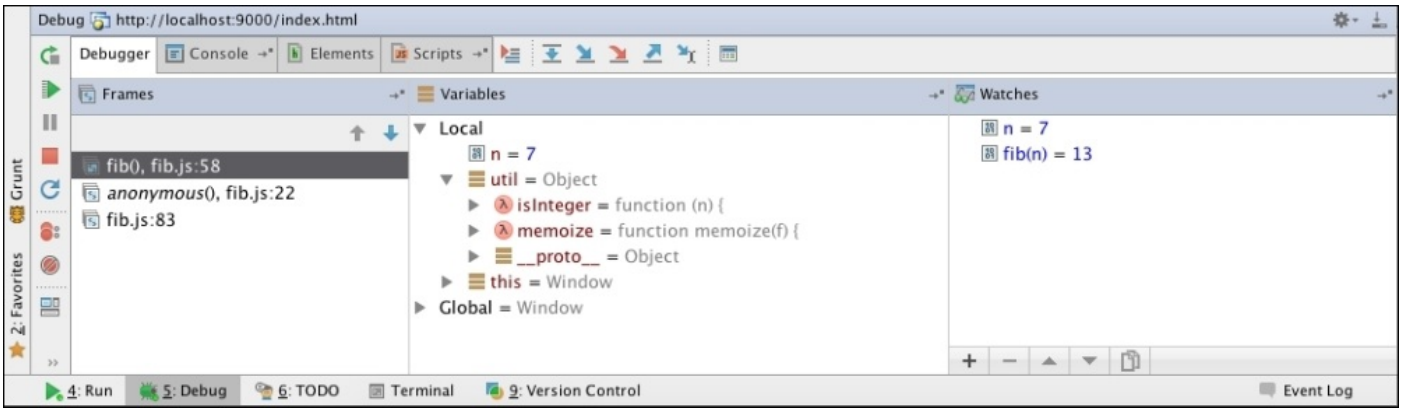
- **Debugger**: This tab shows information relevant to the debugging section like **Frames**, **Variables**, and **Watches**
- **Console**: This tab displays the console output
- **Elements**: The DOM tree is displayed in this tab
- **Scripts**: This tab shows the loaded scripts

Now that we have WebStorm and Chrome linked, we need to refresh the page in Chrome so that it stops at our breakpoints. The browser now shows that our page has paused inside the debugger:



The WebStorm IDE will now jump to the line that the execution has been paused, and display the relevant information in the **Debugger** section. The **Frames** panel will show the call stack that leads to the current point. The **Variables** panel allows you to examine the value stored in your application, on the local scope as well as the global one. In the **Watches** panel, we can evaluate various variables and expressions in the current context. The information displayed in these panels is updated with each step throughout the

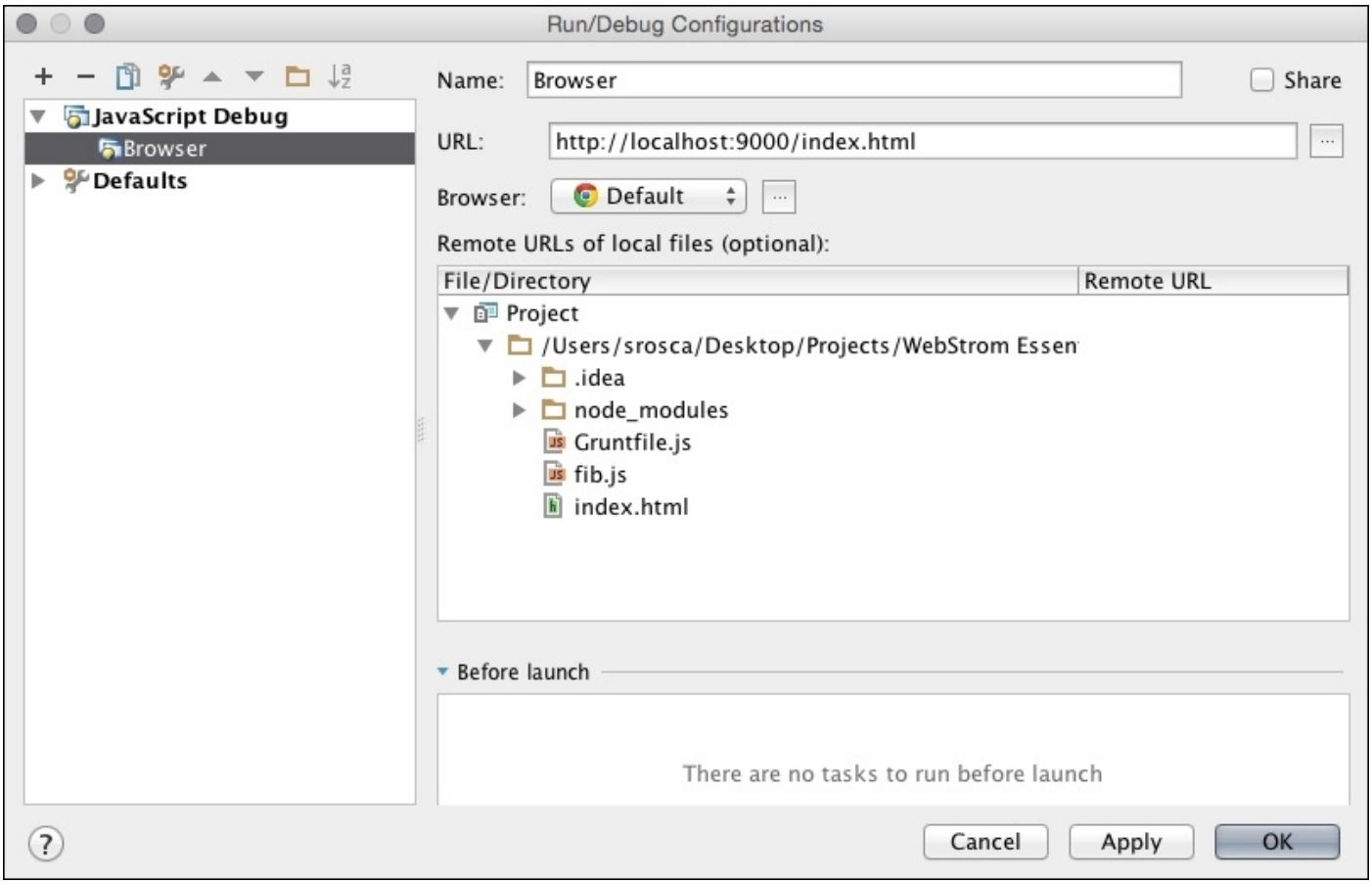
application.



While the execution is paused, we can navigate through the program execution. Navigation can be controlled from the **Run** menu, or by using the following shortcuts:

- **Step over (F8)**: Step to the next line in the current file
- **Step into (F7)**: Step to the next executed line, which can be in a separate file
- **Step out (Shift + F8)**: Step to the first executed line after returning from the current method
- **Run to cursor (Alt + F9)**: Run to the line where the caret is located
- **Resume program (F9)**: Resume program execution

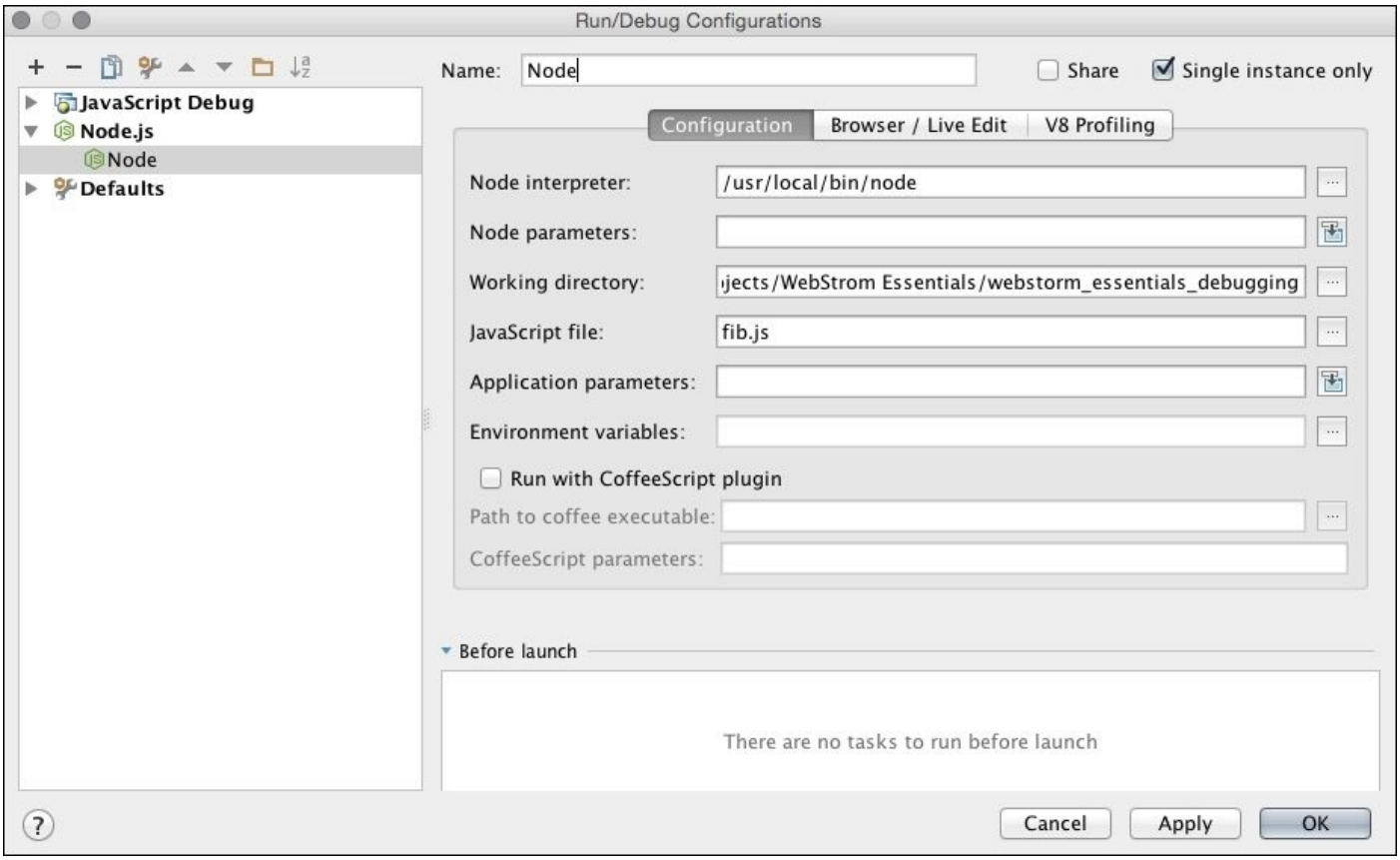
While temporary debug sessions are useful for quick debugging, sometimes we need to save the debug run as a configuration so that it is easier to run. Go to **Run | Edit Configuration** to open the **Run/Debug Configuration** screen and click on the plus sign. Then select **JavaScript Debug**. Set a name for your configuration, enter the URL: `http://localhost:9000/index.html`, and then click on **OK**.



This creates a configuration that can be easily run by selecting the debug icon from the run toolbar, as seen in the following screenshot:

Once we run this, the debugging session is controlled exactly like the previous section.

We can also debug our code inside Node.js, so we need to create a debug configuration for the Node.js environment. To do that, select **Node.js** from the **Run/Debug Configuration** screen, and then select the JavaScript file that you want to debug.



Now we can run the debugging session against the node environment. The rest of the display and controls remain the same as in the previous examples.

Debugging is one of the most powerful ways to understand the execution of the code, and WebStorm's ability to debug the code directly in the environment in which it is created, the IDE, is an extremely valuable benefit.

Summary

In this chapter, you have learned to analyze and debug your code. You installed and configured some of the most-used code quality tools and used them on your code. You have also learned to debug your code and find problems quickly. These are the basic steps that will allow you to better understand how your code is being executed, and by following some simple rules, you can be sure that the code is performing and looking great.

In the next chapter, we are going to see how WebStorm can help you in testing your code so that we are sure that whatever changes we make will not break our code. We are going to dive into some of the most-used development practices of the moment: TDD and BDD.

Chapter 8. Testing Your Applications

In the previous chapter, we went through the process of analyzing and debugging our code. We learned to use the Code Inspector, Code Style checker, and the Code Quality Tools.

In this chapter, we are going to see how WebStorm helps us in testing our code for ensuring that it is performs as expected. We will learn how to configure the test runners, run tests inside the IDE, and to use the following frameworks:

- Karma
- Jasmine
- Node Unit
- Mocha
- Cucumber.js
- Wallaby.js

Note

Before we start, please note that the processes explained in this chapter might seem to be repetitive. However, this will not be the case in a production environment, since you usually use only one framework. In WebStorm, the differences, when working with multiple testing frameworks, are mostly at the syntax level.

Karma

Karma is a test runner created by the AngularJS team that helps us run out tests against several browsers. It starts the browsers that you select, loads the file you specify, and reports the results from your tests. Karma supports multiple-tests frameworks, so you can write your tests in Jasmine, Mocha, and so on.

Before you start, you need to install the command interface globally so that you can run Karma directly without having to go to the `node_modules` folder. In the terminal, run the following command:

```
npm install -g karma-cli
```

Once you have the CLI installed globally, download the sample chapter from GitHub. So, create a new project using the Git repository at https://github.com/srosca/webstorm_essentials_testing.git as a source.

After downloading the project, you need to install Karma as a development dependency:

```
npm install karma --save-dev
```

Now you can initialize Karma in the project. To do that, run the following command in the terminal:

```
karma init
```

You will be asked several questions for configuring the project. Answer them using the following options:

- **Which testing framework do you want to use?**

  Jasmine—to use Jasmine as the testing framework

- **Do you want to use Require.js?**

  no—since we don't want to use `Require.js`

- **Do you want to capture any browsers automatically?**

  Chrome—as we want to use Chrome from the beginning enter an empty string to move to the succeeding questions

- **What is the location of your source and test files?**

  fib.js—to test our file

  test/jasmine/*.spec.js—we are going to keep all our tests in the `test/jasmine` folder with `spec` as the name

  You have to again enter an empty string to move to the next question

- **Should any of the files included by the previous patterns be excluded?**

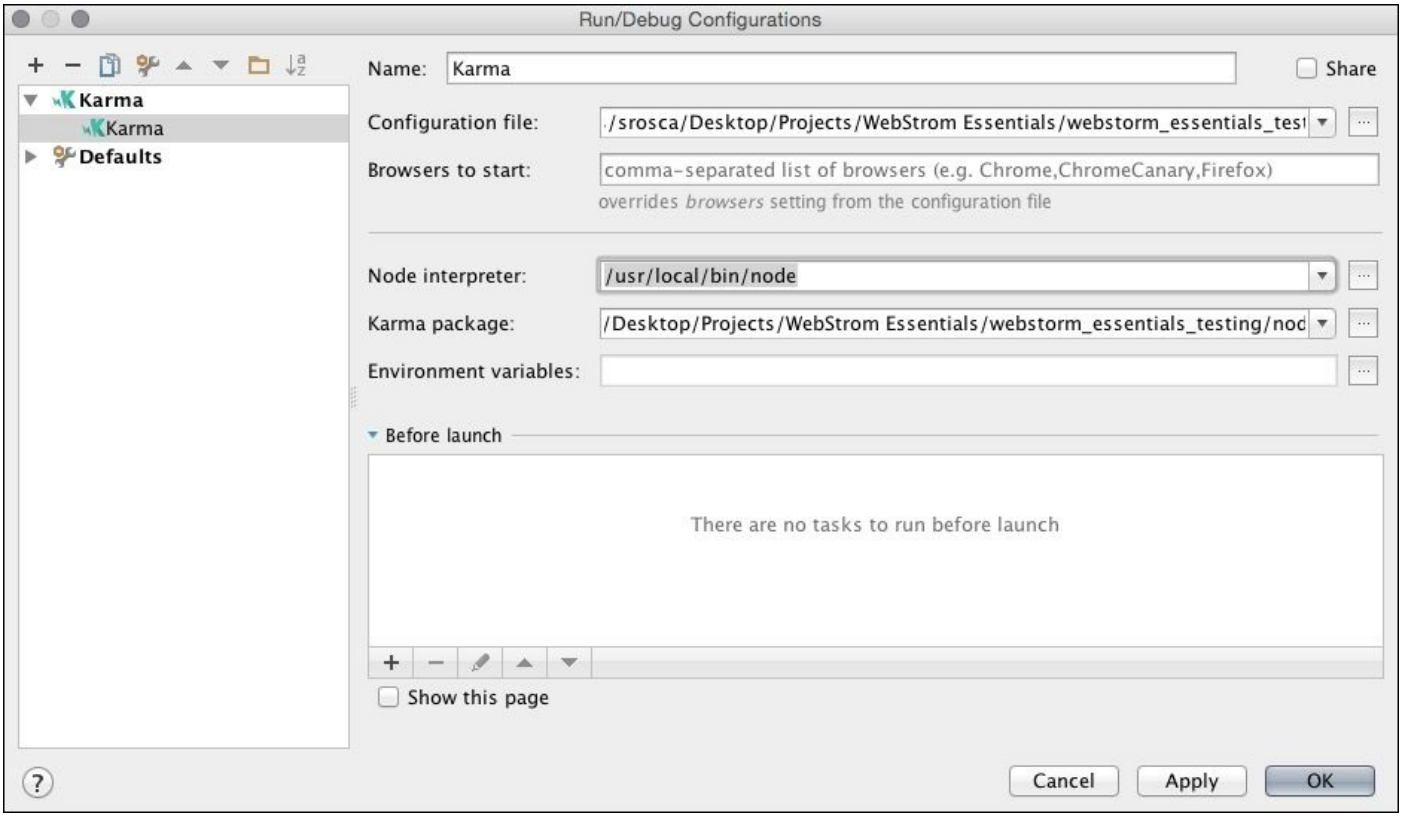
  Leave this option blank

- **Do you want Karma to watch all the files and run the tests on change?**

yes—as we want to watch for changes and run the test automatically

Once you're done with this, you will have created the `karma.conf` file that keeps the settings for your project. Moreover, based on the selections that we make, Karma installs the required dependencies: `karma-chrome-launcher`, `karma-jasmine`, and `jasmine-core`.

You can now create a **Run/Debug Configuration** to run Karma inside WebStorm. Navigate to **Run** | **Edit Configurations**, and select **Karma** from the **Add New Configuration** menu accessible from the + icon or by pressing *Ctrl* + *N*.



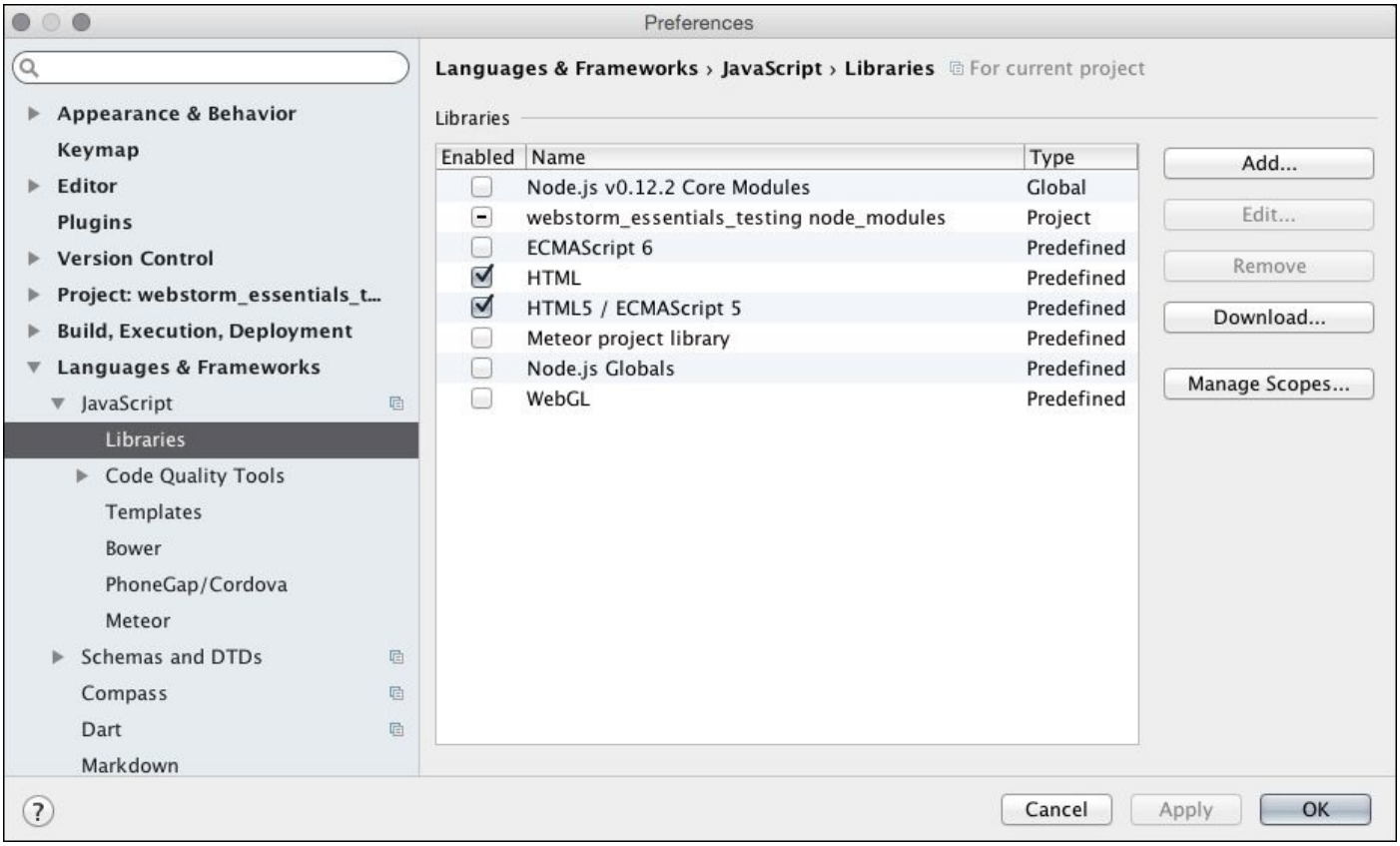
In the configuration screen (shown in the preceding screenshot), you need to select the path to the `karma.config.js` file, node, and Karma. WebStorm picks these up automatically.

Now we can start writing the tests for our code. One of the frameworks that we are going to look into first is Jasmine.

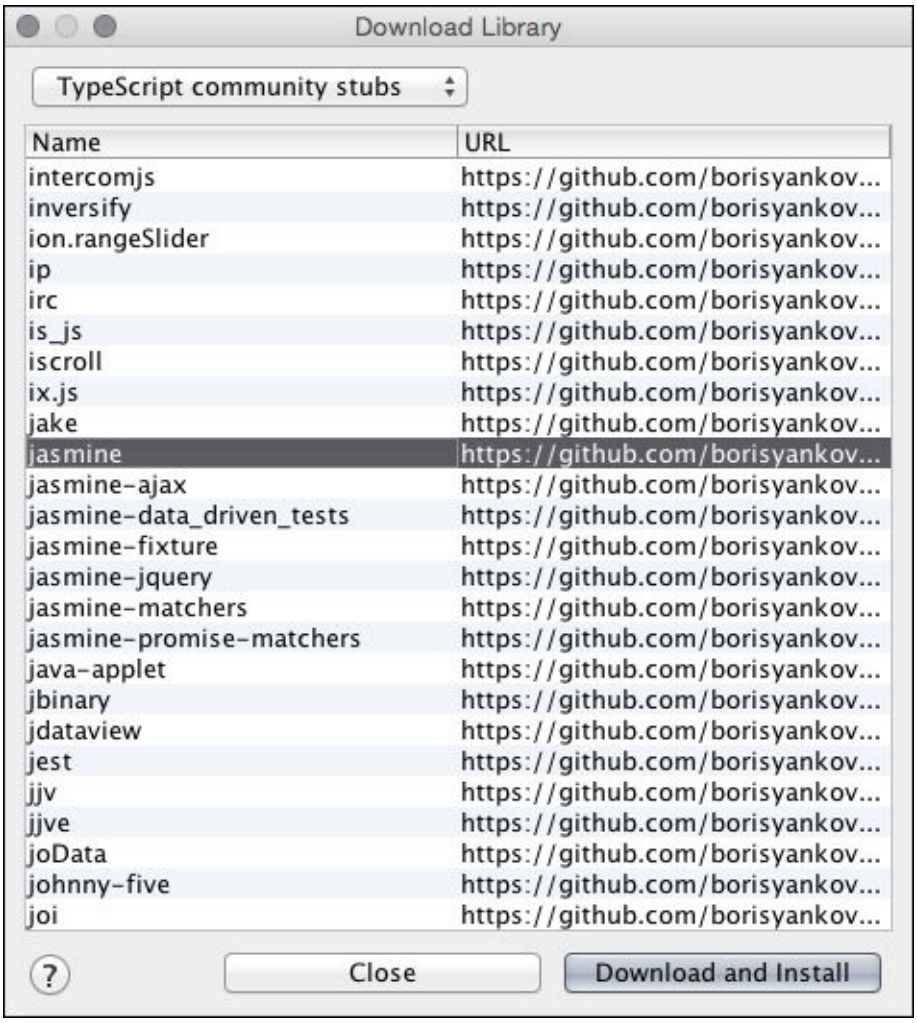
Jasmine

Jasmine is a behavior-driven development framework for testing JavaScript code. It is designed to run on any JavaScript platform, does not depend on any other framework or the DOM, and has an easy-to-read obvious syntax.

Before we start, it will help to install the Jasmine JavaScript library for WebStorm. We can use autocomplete and syntax highlight. Go to the **Preferences | JavaScript | Libraries** screen, and select **Download**.



In the new screen, select **TypeScript community stubs** and search for **jasmine**, and then select **Download and Install**.



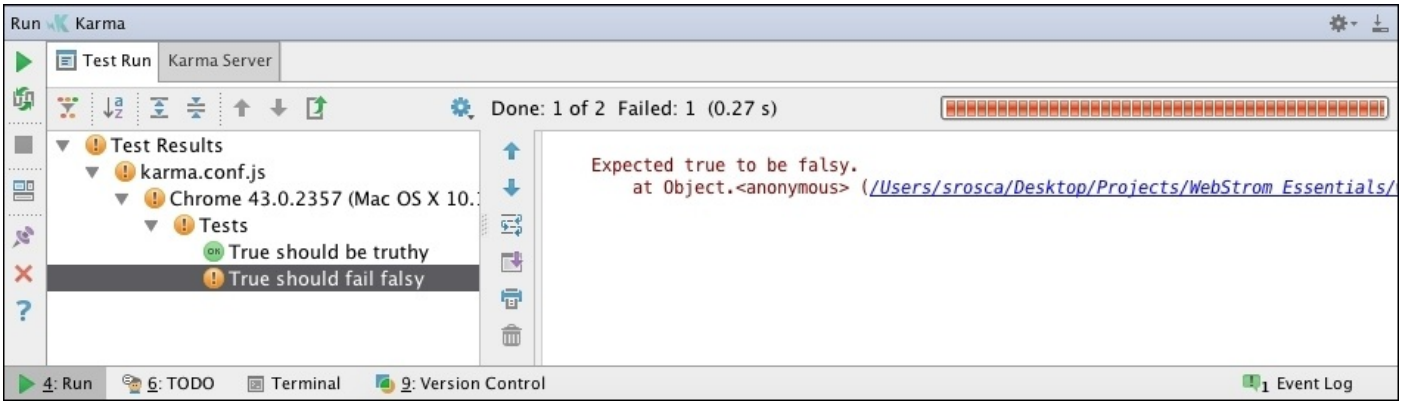
Now that you have the library set up, you can start. First you need to create the `test` folder, a `jasmine` folder inside the `test` folder, and a `fib.spec.js` file in the `jasmine` folder. As a convention, the tests use the tested file name followed by `.spec.js`. In practice, the test files are placed directly inside the `test` folder, but we have created the `jasmine` folder so that it is easier to use multiple frameworks.

After creating the file, you need to fill it up with the following code:

```
(function () {
    'use strict';

    describe('Tests', function () {

        it('True should be truthy', function () {
           expect(true).toBeTruthy();
        });

        it('True should fail falsy', function () {
            expect(true).toBeFalsy();
        });
    });
})();
```

This simple code will do some simple tests to check that everything is okay. You can now

start Karma from the **Run** menu, the toolbar, or by using the shortcut, *Shift* + *W*. This will start Karma and display the results in the run section.



These tests are just a simple example to make sure that everything is set up correctly. If you are able to see the same result like the preceding screenshot, we can now move to the next step, testing our Fibonacci function. While writing the test, it is a good practice to run the tests in continuous mode. We can do that by toggling auto run from the run section  . This is the equivalent of the watch mode, and it will run the tests automatically whenever we change the files.

You will first have to delete the demo test that we did, and then get the `fib` function from the global object. Finally, you will test the result for some predefined values. You will also check if the recursion works okay by verifying that the current result is equal to the sum of the previous two results. The final code should look like this:

```
(function () {
    'use strict';

    var fib = window.fib;

    describe('Fibonacci', function () {

        it('Should return the term at the given position', function () {
            expect(fib(0)).toEqual(0);
            expect(fib(1)).toEqual(1);
            expect(fib(2)).toEqual(1);
            expect(fib(3)).toEqual(2);
            expect(fib(4)).toEqual(3);
            expect(fib(8)).toEqual(21);
            expect(fib(9)).toEqual(34);
            expect(fib(51)).toEqual(20365011074);
        });

        it('Should return the value as the sum of previous values',
function(){
            expect(fib(0) + fib(1)).toEqual(fib(2));
            expect(fib(1) + fib(2)).toEqual(fib(3));
            expect(fib(10) + fib(11)).toEqual(fib(12));
            expect(fib(2500) + fib(2499)).toEqual(fib(2501));
```

```
        });

    });
})();
```

If you have toggled the auto run mode, then your run section should display the result as you complete them. If everything goes well, the entire tests should pass.

In this example, you have created a test to check that your function calculates the value correctly with the help of Jasmine and Karma. In the next section, we are going to use another testing framework: Nodeunit.

Nodeunit

Nodeunit is a framework for testing the code for Node.js. Its main focus is to create an easy way to test the code written for node. Some of its main features are as follows:

- Simple to use
- Ability to export tests from a module
- Works with Node.js and in the browser
- Allows the use of mocks and stubs

Before writing the tests, you need to install Nodeunit as a development dependency. Run the following command in the terminal:

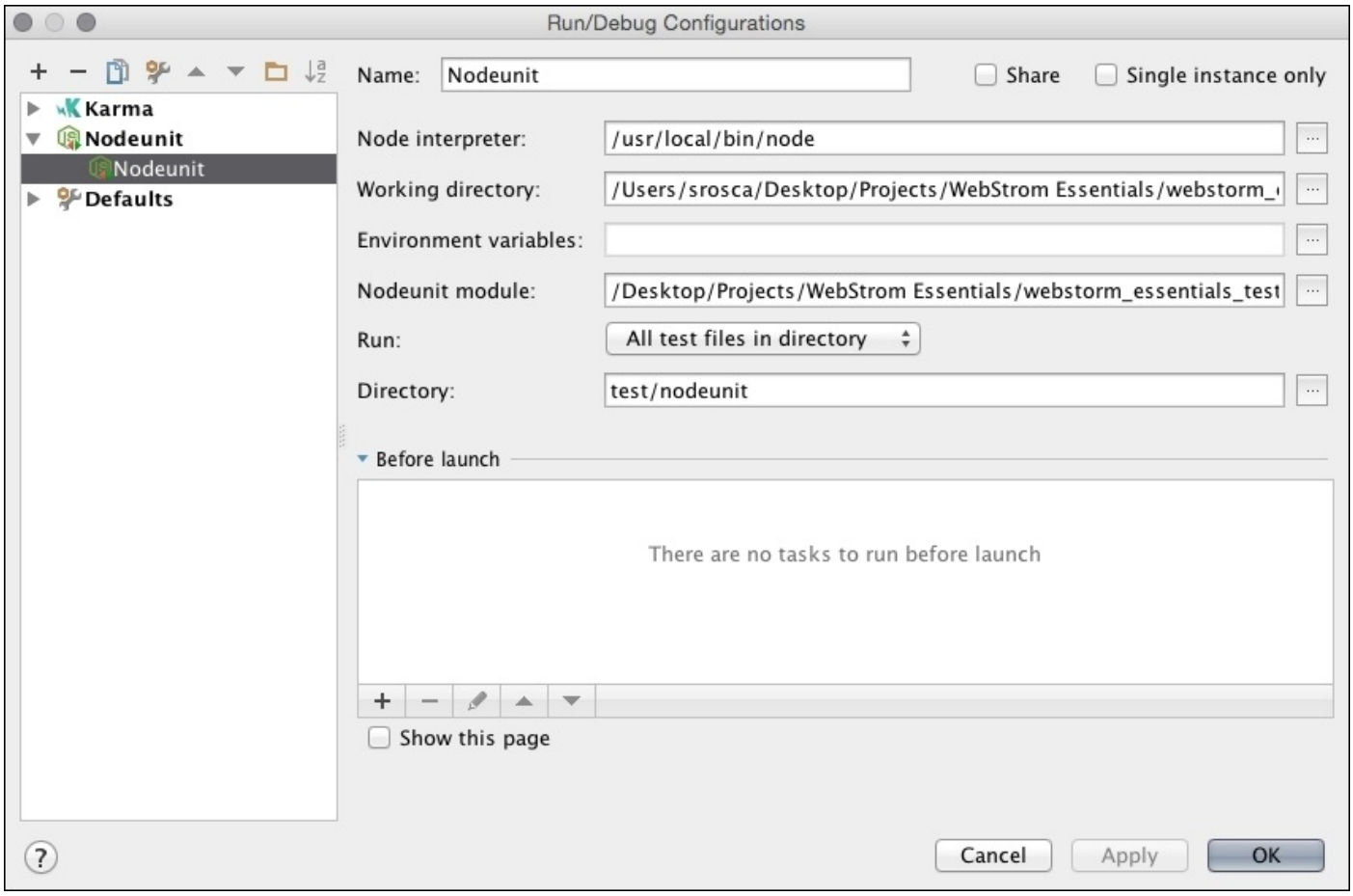
```
npm install nodeunit --save-dev
```

Once the package is installed, create a `fib.spec.js` file inside the new `nodeunit` folder in the test . After the file is created, add in the following code:

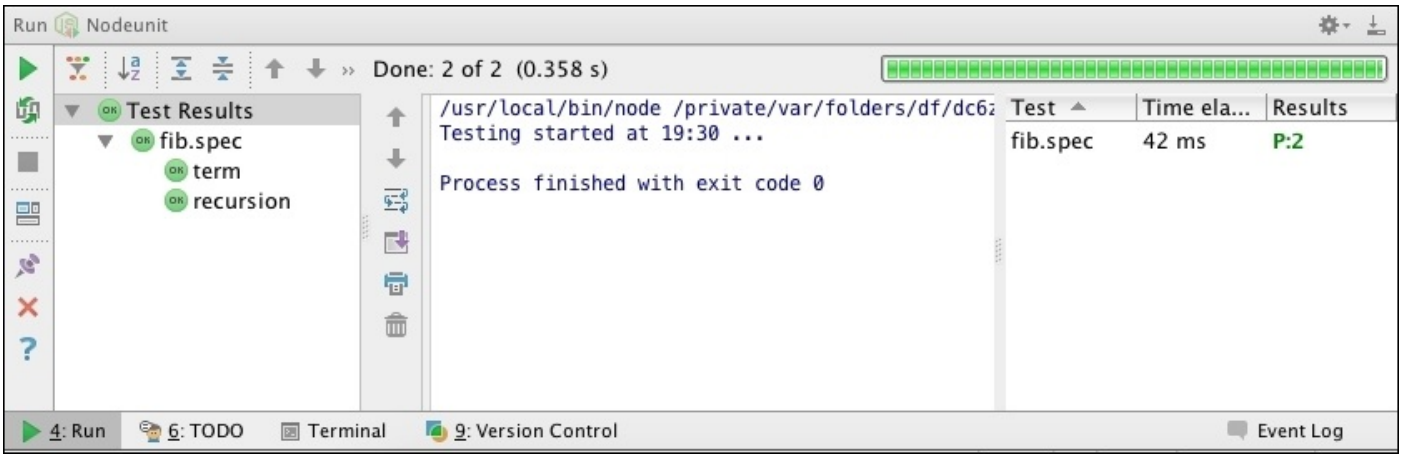
```
(function () {
    'use strict';

    var fibModule = require('./../../fib.js');
    var fib = fibModule.fib;

    exports.term = function (test) {
        test.equal(fib(0),0, 'Fibonacci 0 is 0');
        test.equal(fib(1),1, 'Fibonacci 1 is 1');
        test.equal(fib(2),1, 'Fibonacci 2 is 1');
        test.equal(fib(3),2, 'Fibonacci 3 is 2');
        test.equal(fib(4),3, 'Fibonacci 4 is 3');
        test.equal(fib(8),21, 'Fibonacci 8 is 21');
        test.equal(fib(9),34, 'Fibonacci 9 is 34');
        test.equal(fib(51),20365011074, 'Fibonacci 51 is 20365011074');
        test.done();
    };

    exports.recursion = function (test) {
        test.equal(fib(2),fib(0) + fib(1), 'Fibonacci 2 is  fib 0 + fib
1');
        test.equal(fib(3),fib(1) + fib(2), 'Fibonacci 3 is  fib 1 + fib
2');
        test.equal(fib(12),fib(10) + fib(11), 'Fibonacci 12 is fib 10 + fib
11');        test.equal(fib(2501),fib(2499) + fib(2500), 'Fibonacci 2501 is
fib 2499 + fib 2500');

        test.done();
    };
})();
```

The tests are similar to the ones in the Jasmine section, but are written with the Nodeunit syntax. To run them, you need to create another run configuration but by selecting Nodeunit as a starting point this time. Open the configuration screen from the **Run** | **Edit Configurations** menu, and fill in the necessary settings. Use the installed package as the

Nodeunit module, **All files in directory** as a **Run**, and `test/nodeunit` as the **Directory**.



Once you have created the configuration, you can run the tests in the same way as the previous section, from either the toolbar or the **Run** menu. This will display the run section with the test results, as seen in the following screenshot:



As you can see, the workflow for creating and running tests is very similar between frameworks, the only difference being in the syntax that we write. Next, we are going to see another framework: Mocha.

Mocha

Mocha is a framework for testing code for both Node.js and the browser. It is a more complex one with more features. It allows the user to write the tests using several syntaxes like BDD, TDD, exports, and so on. The tests in Mocha are run serially, which allows for a more flexible and accurate reporting, and the exceptions are correctly mapped to the test case.

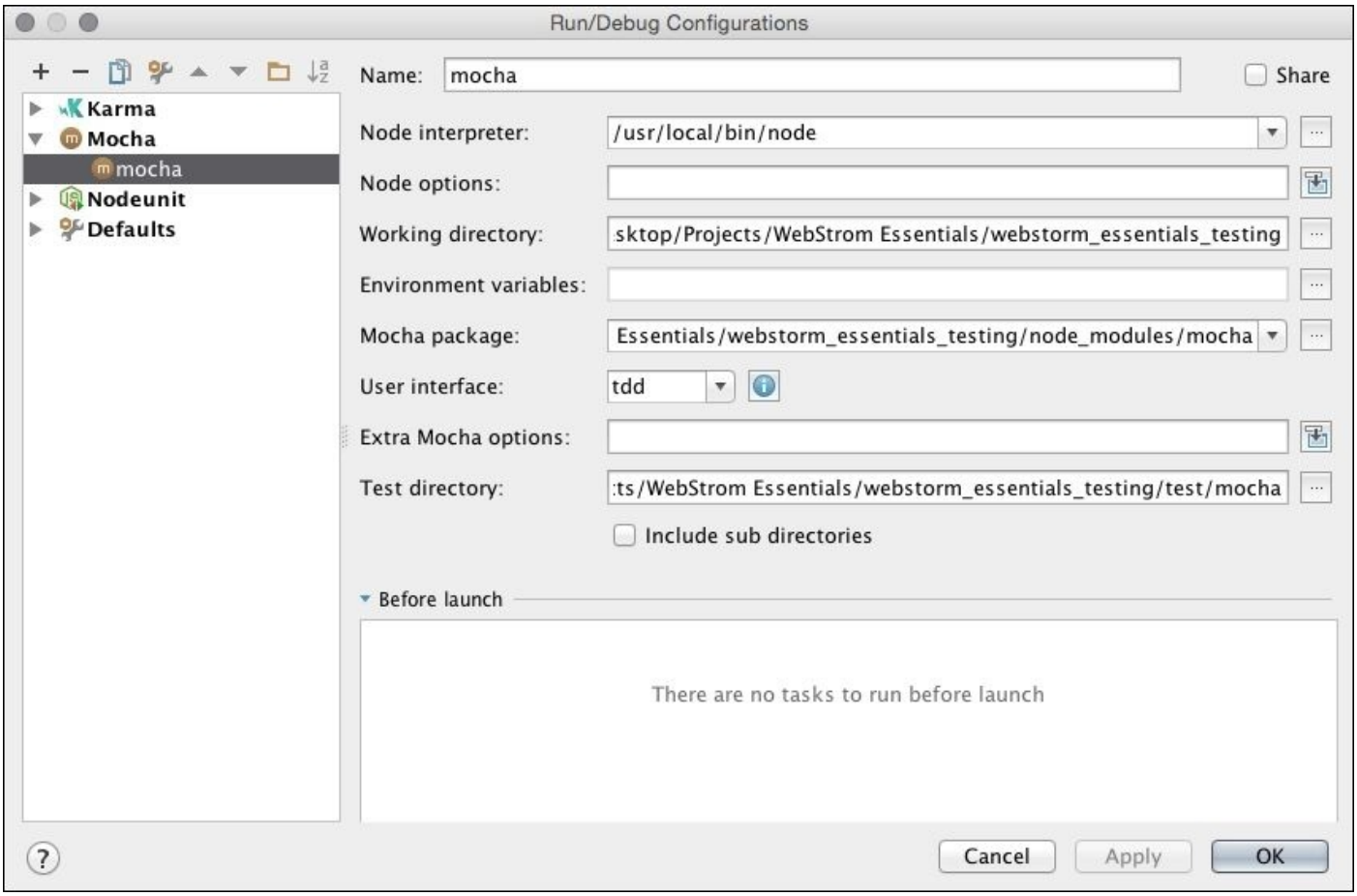
Again, before we write the tests, we need to install Mocha as a development dependency by running the following command in the terminal:

```
npm install mocha --save-dev
```

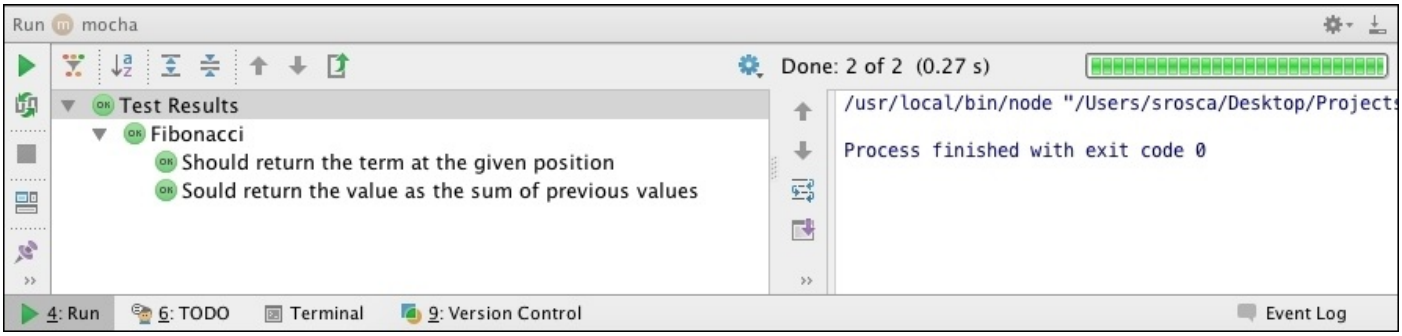
Now, create a `fib.spec.js` file in a new `mocha` folder in `test`. Once the file is created, you need to fill the following tests:

```
(function () {
    'use strict';
    var assert = require("assert");

    var fibModule = require('./../../fib.js');
    var fib = fibModule.fib;

    suite('Fibonacci', function () {test('Should return the term at the
given position', function () {
            assert.equal(fib(0), 0);
            assert.equal(fib(1),1);
            assert.equal(fib(2),1);
            assert.equal(fib(3),2);
            assert.equal(fib(4),3);
            assert.equal(fib(8),21);
            assert.equal(fib(9),34);
            assert.equal(fib(51),20365011074);
        });

        test('Should return the value as the sum of previous values',
function () {
            assert.equal(fib(0) + fib(1), fib(2));
            assert.equal(fib(1) + fib(2), fib(3));
            assert.equal(fib(10) + fib(11), fib(12));
            assert.equal(fib(2500) + fib(2499), fib(2501));
        });
    });

})();
```

As you can see, the differences are again only at the syntax level. The logic of the tests remains the same. Now that you have the test created, create a run configuration, selecting **Mocha** as the template this time.



Select the **Mocha** package, select **tdd** as a user interface, the test directory and then save the configuration, as seen in the preceding screenshot. This allows us to run the Mocha test and see the results in the run section.



Next, we are going to focus on a framework that tries to use a more human-readable way to write the test.

Cucumber.js

Cucumber.js is a BDD framework that runs in Node.js and the browser. It uses a simple-to-understand syntax—the Gherkin language—that is described as a business readable, domain-specific language.

The test that you will write in Cucumber comprises two types of files:

- Feature files written in Gherkin
- Support files that are written in JavaScript or CoffeeScript

One of the first things to do before writing the test is to install Cucumber.js as a development dependency by running the following command:

```
npm install cucumber -save-dev
```

Once the package is installed, you need to create the folder structure; so, create a `cucumber` folder inside the `test` one. Inside the `cucumber` folder, create the features file, `fib.feature`, and fill it with the following code:

```
Feature: User can calculate the nth term in the Fibonacci seriesAs a user
  I want to calculate the nth term in the Fibonacci series
  So that I know what the value is

  Scenario Outline: Value calculated for <number>
    Given I have a <number> as n
    When I pass n to the fib function
    Then It calculate the <result> as value of the nth term

    Examples:
      |number |result |
      |0      |0      |
      |1      |1      |
      |2      |1      |
      |3      |2      |
      |4      |3      |
      |8      |21     |
      |9      |34     |
      |51     |20365011074 |
```

As you can see, the Gherkin syntax focuses on the user story and not on the technical implementation. This is handled in the support files that we are going to create. These files will be placed in the `step_definitions` folder that you need to create inside the `cucumber` folder. In this folder, create the `fib.steps.js` file, and fill it with the following code:

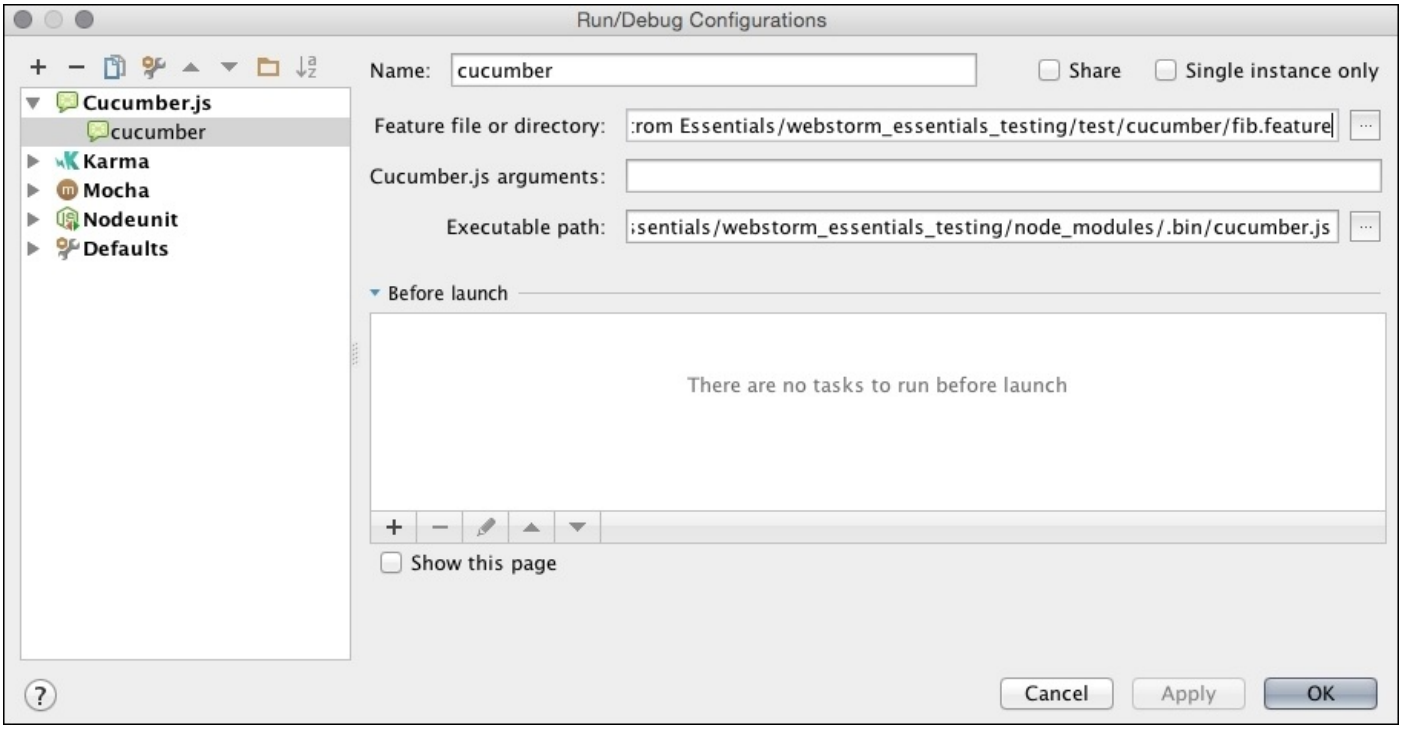
```
'use strict';
var assert = require('assert');

var fibModule = require(process.cwd() + '/../fib.js');
var fib = fibModule.fib;
var n, result;

module.exports = function() {
```

```
    this.Given(/^I have a (\d+) as n$/, function(number, callback)
{
        n = parseInt(number, 10);
        callback();
    });

    this.When(/^I pass n to the fib function$/, function(callback)
{
        result = fib(n);
        callback();
    });

    this.Then(/^It calculate the (\d+) as value of the nth term$/,
function(value , callback) {
        assert.equal(result, value, 'fib of ' + n +'should be ' + value );
        callback();
    });

};
```

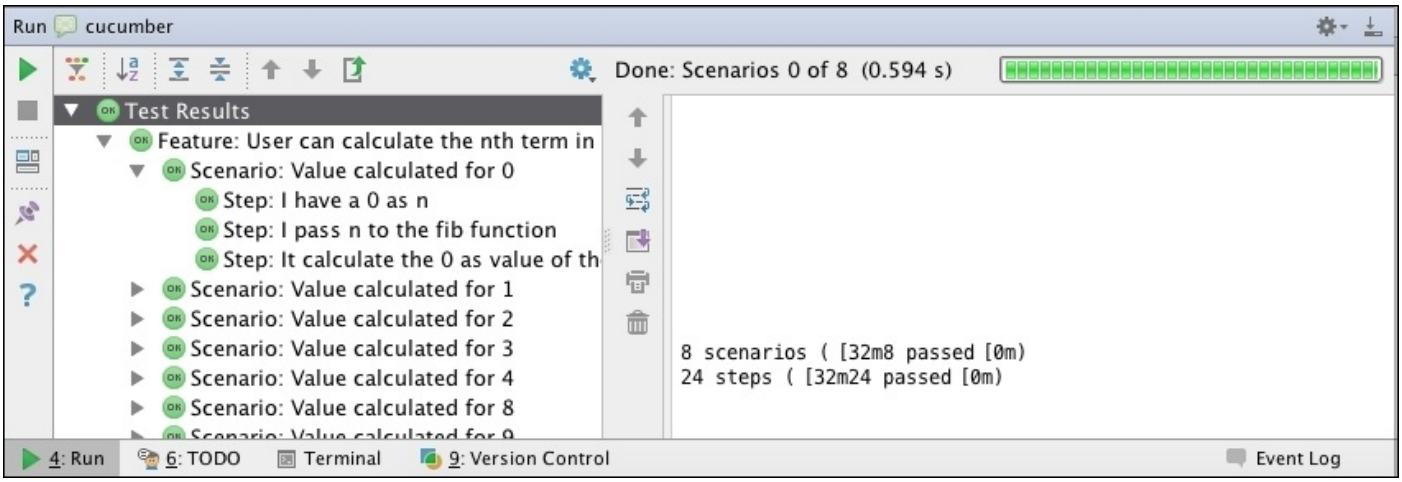
As you can see, in this file, we are translating the steps from the features files into technical implementation.

Now that you have all the files created, you can create a run configuration in the same way as in the previous section, but selecting `Cucumber.js` as the template this time.

You need to fill the **Feature file or directory** with the `fib.feature` file that we have created, and specify the executable path to the `Cucumber.js` file from the `node_modules/.bin/ folder`.



Having performed these steps, you will have created a new run configuration that can be started from the toolbar or the **Run** menu, as seen in the following screenshot:



In the run section, you can see the results of the test run grouped by features and scenarios. As you can see, the cucumber framework focuses more on creating the test in a syntax that is easy to read and translate into business requirements.

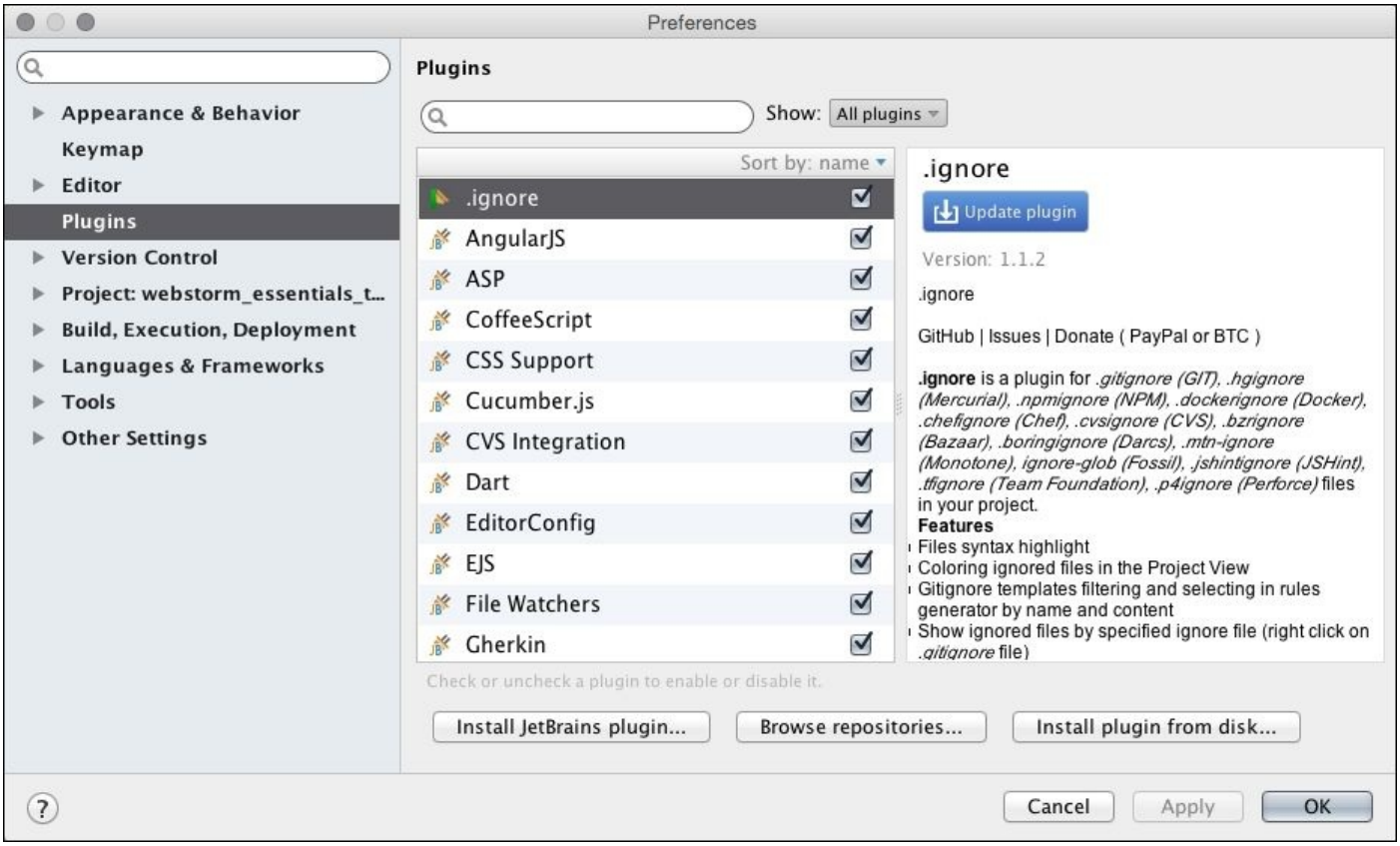
Next, we are going to focus on an intelligent test runner that runs and displays the results in a clever way: directly in your code.

Wallaby.js

Wallaby.js is a test runner that runs your test in an innovative way. The tests are run in a continuous mode and Wallaby.js reports the code coverage and results directly in your code editor as you change the code.

Before using Wallaby.js, you need to download the plugin from the product page at http://wallabyjs.com/.

After downloading the plugin, install it from the **Preferences** | **Plugins** dialog, and select **Install plugin from disk…**, as shown in the following screenshot:



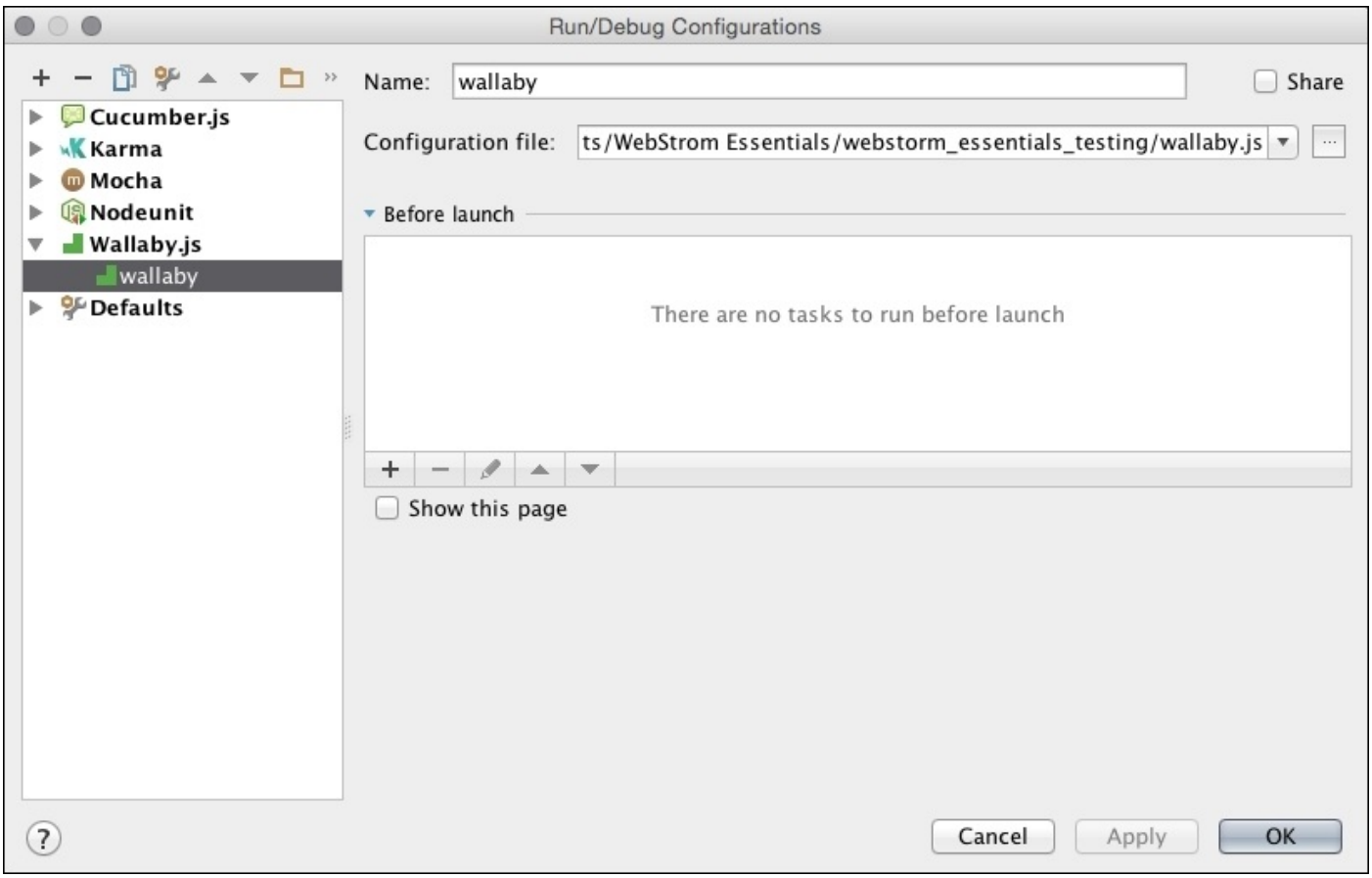
Follow the steps for installing the downloaded plugin, and restart WebStorm. Since Wallaby.js is a paid plugin, after restarting, you will be prompted for the license or for using the trial mode.

Once you have it installed, you need to create a simple configuration file that will specify the files that you are using. Create a `wallaby.js` file in the project root, and fill it with the following code:

```
module.exports = function (wallaby) {
  return {
    files: ['fib.js'],
    tests: ['test/jasmine/*.spec.js']
  };
};
```

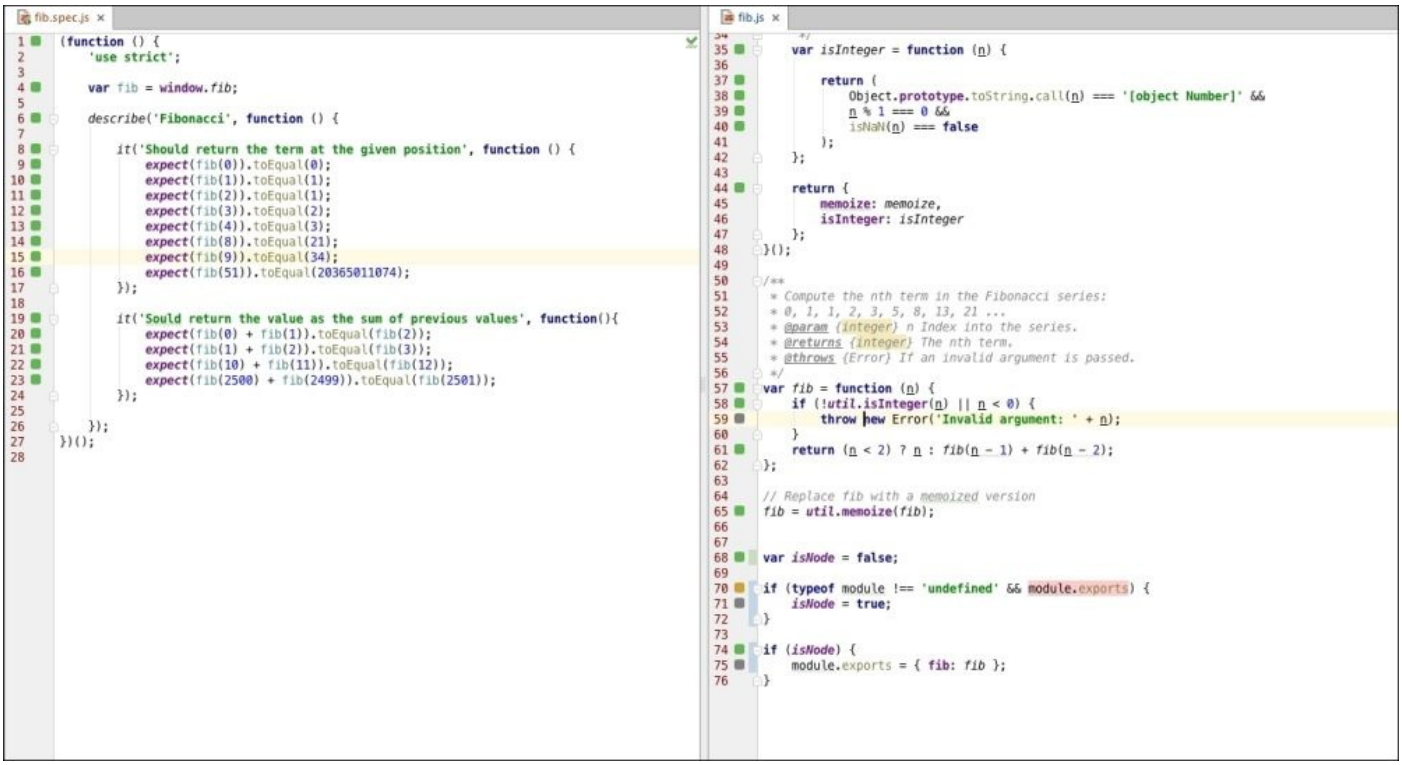
You are going to use the Jasmine tests that we have already defined in the previous

section. Now create a run configuration that will run **wallaby**. Using the known steps, create a configuration based on the **Wallaby.js** template:

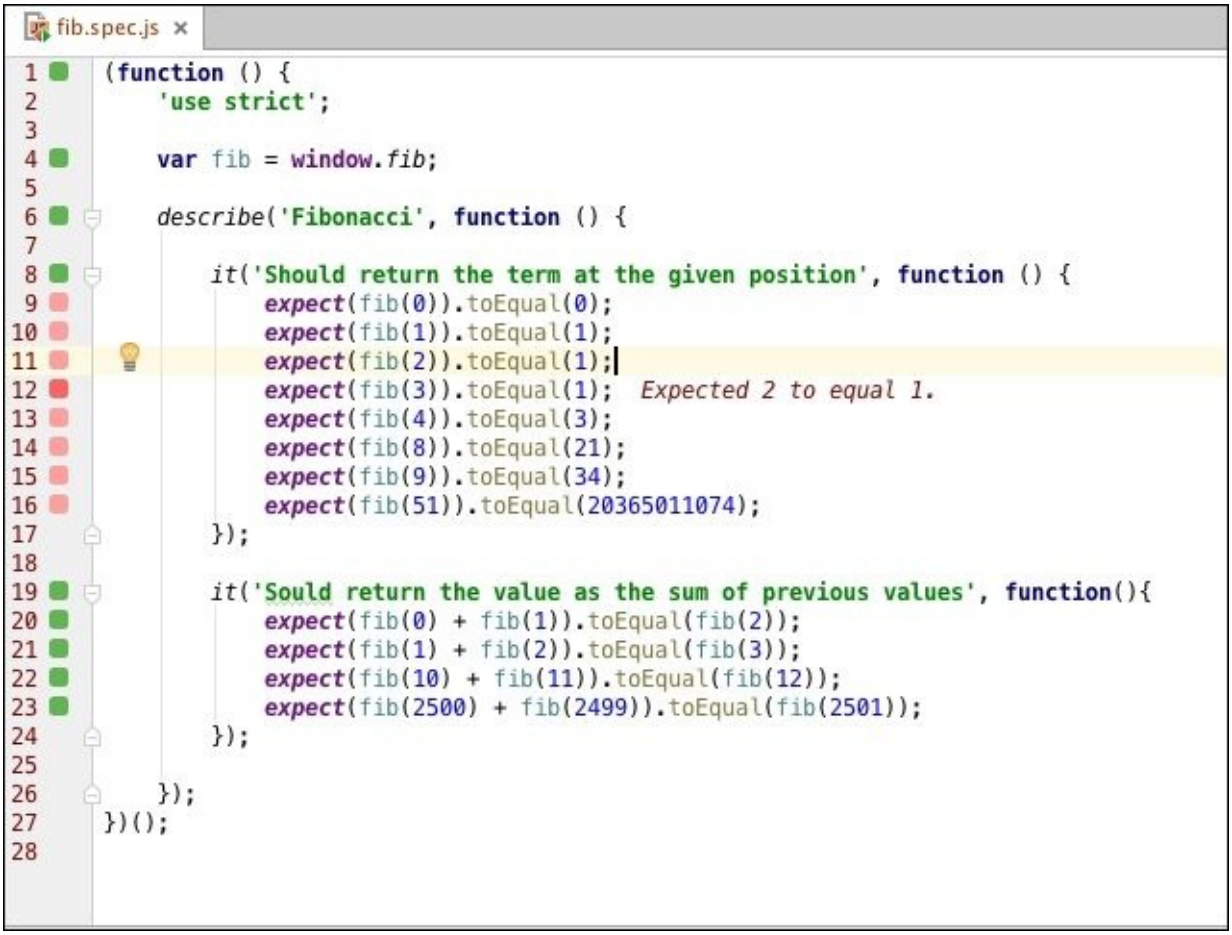


You need to specify the `wallaby.js` configuration file that you created in the previous step.

When you run the project from the toolbar or from the **Run** menu, you will see some details in the run section, but this time the results and code coverage will be displayed as colored squares directly in your test files and source code.



Moreover, if you change one of your tests to fail, the failure will be displayed in line with the code so that you can easily see what the problem is.



As you can see, one of the big advantages that come with Wallaby.js is that you can see your results quickly, directly inside the code. This way, you can make changes quickly, without the need to jump between tools.

Summary

In this chapter, we focused on testing our code. We worked with some of the popular test runners and testing frameworks so that we can easily see the problems in our code.

In the next and final chapter, we are going to see some of WebStorm’s features that will boost our productivity: Live Edit, TODO facility, Emmet, and others.

Chapter 9. Getting to Know Additional yet Powerful Features

In the previous chapter, we have seen how WebStorm works together with test runners and testing frameworks to ensure that our code is tested.

In this final chapter, we are going to focus on some of WebStorm's power features that help us boost our productivity and developer experience.

We will cover the following topics in this chapter:

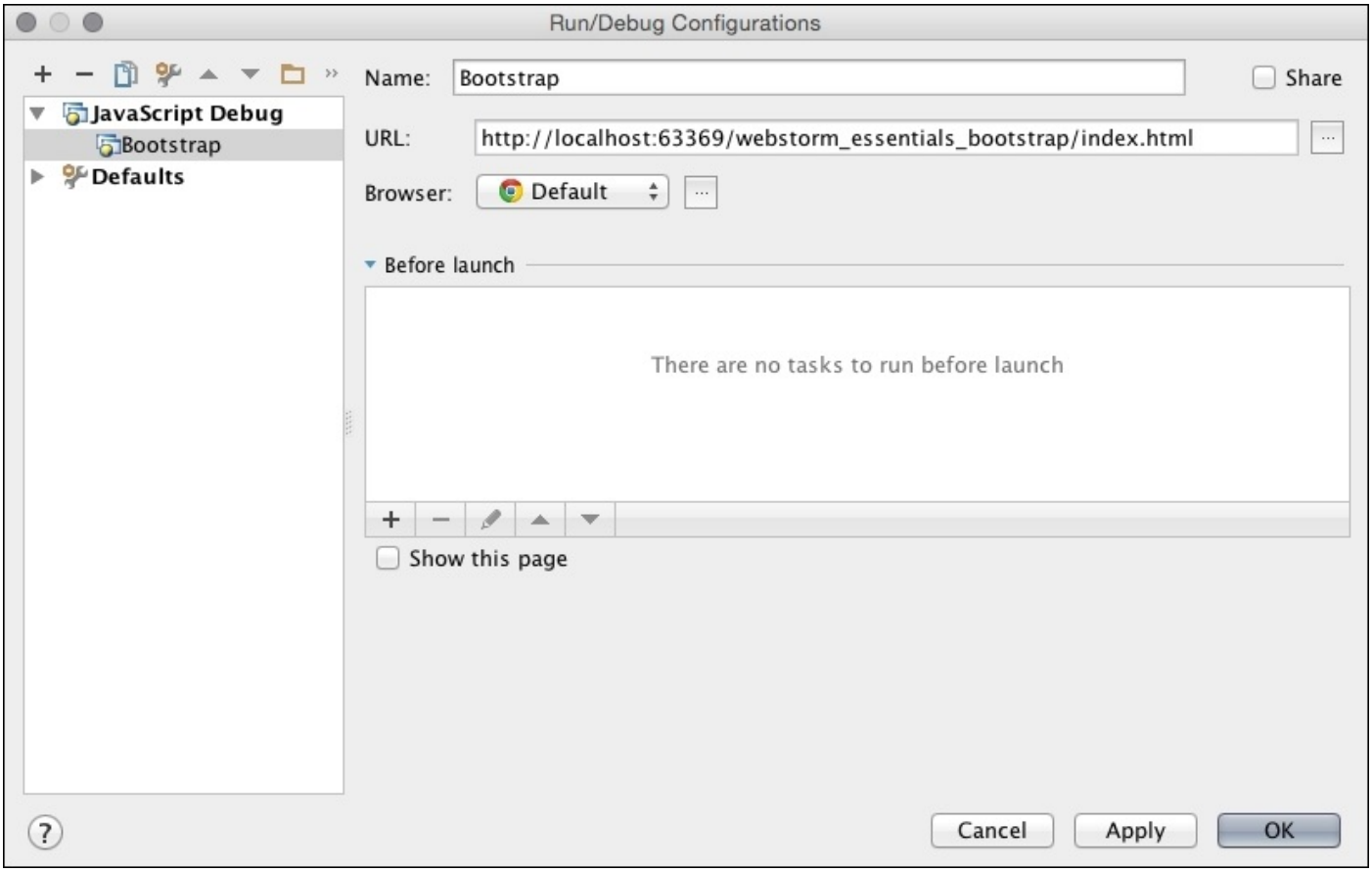
- Using the Live Edit mode
- Working with Emmet
- The TODO facility
- The Differences Viewer
- Tracking Local History

Using the Live Edit mode

While you are in a JavaScript debug session, WebStorm allows you to see all the changes that you make to any HTML file or any file that generates HTML, CSS, or JavaScript . When you are in this mode, you can also see the page structure and scripts in the **Elements** tab of the **Debug** section.

Before we use the Live Edit mode, we need to make sure that the plugin is activated in the plugins settings page, and that the WebStorm browser extension is installed and activated. We've already used this in the debugging section.

For this section, we are going to use the project that we created in the Bootstrap example. So open the example, or create a new project from the GitHub repository at https://github.com/srosca/webstorm_essentials_bootstrap.git. Once you have all the necessary code, open the **Run/Debug Configurations** dialog from **Run** | **Edit Configurations…**, as shown in the following screenshot. Select JavaScript as the template, Bootstrap as the name, and `http://localhost:63369/webstorm_essentials_bootstrap/index.html` as the URL.

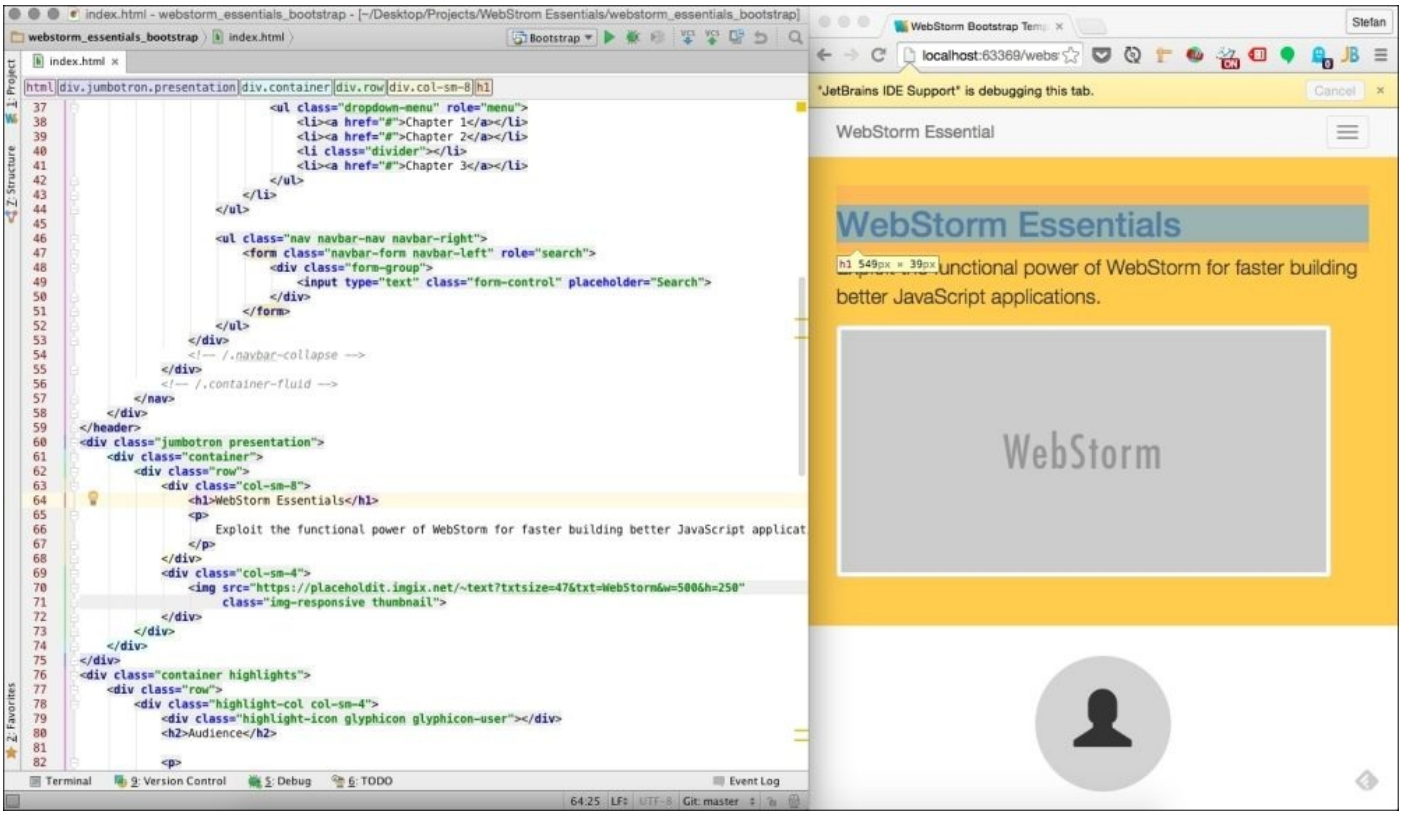


Once you have set the run configuration, you can start a debugging session from the **Run** menu or the toolbar. With this session started, you have access to multiple tools that will help with the development of your project.

All the code in the HTML file will be highlighted in the browser when you change your

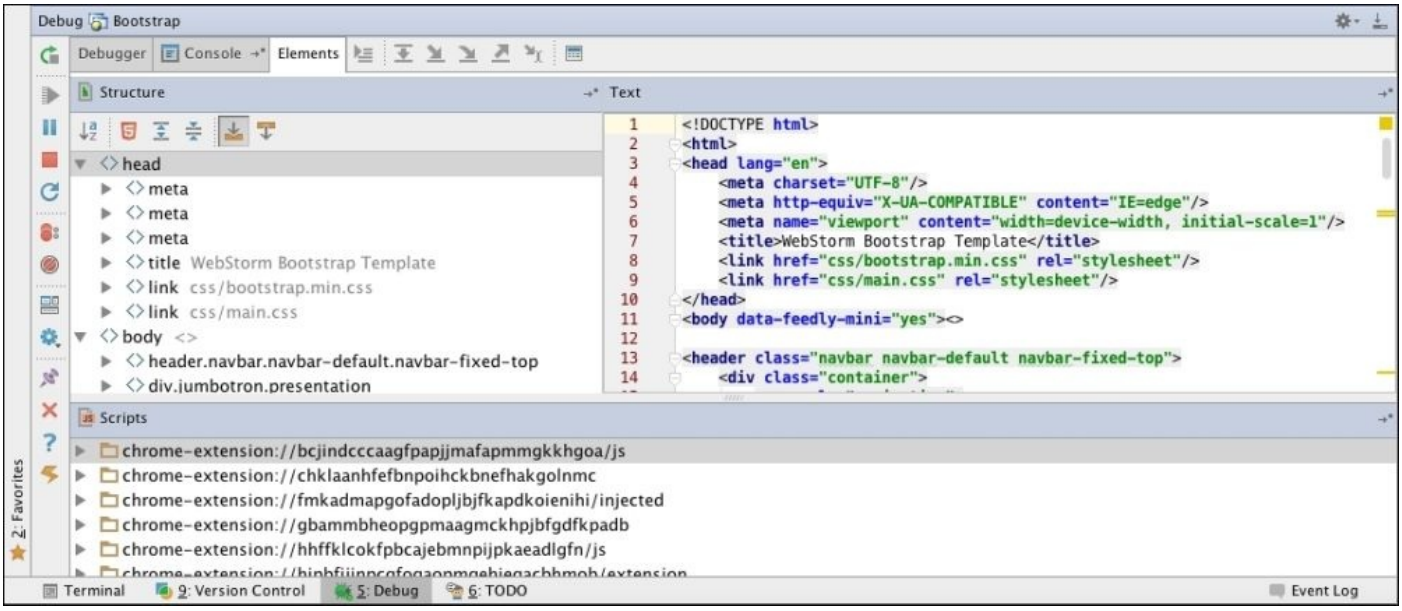
caret location, or when we shift click an element.

The next image shows the usual setup for the Live Edit mode, where you keep both the editor window and the browser opens at the same time, and as you edit your code, the changes are reflected immediately.



Now if we change the HTML file, all the changes will be reflected immediately in the browser. WebStorm knows how to refresh all the changes we are making to HTML, CSS, JavaScript files, or any files that generate changes.

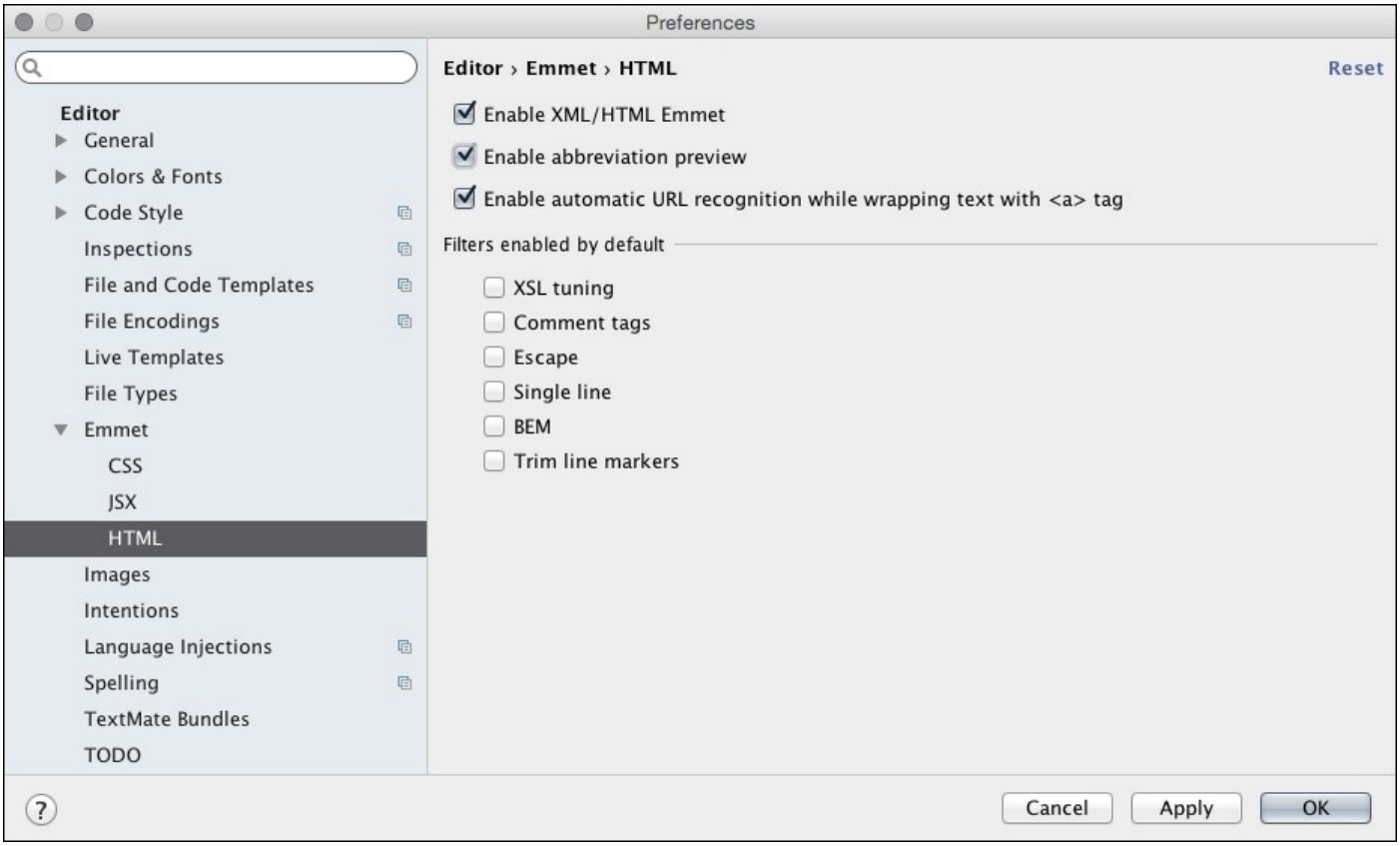
In the Live Edit mode, you can also find an overview of the page structure and scripts, loaded in the **Debug** section in the **Elements** Tab.



The Live Edit mode is really useful from the developer’s point of view, since he/she can quickly see how the changes affect the page, and is thus able to quickly debug and prototype the pages.

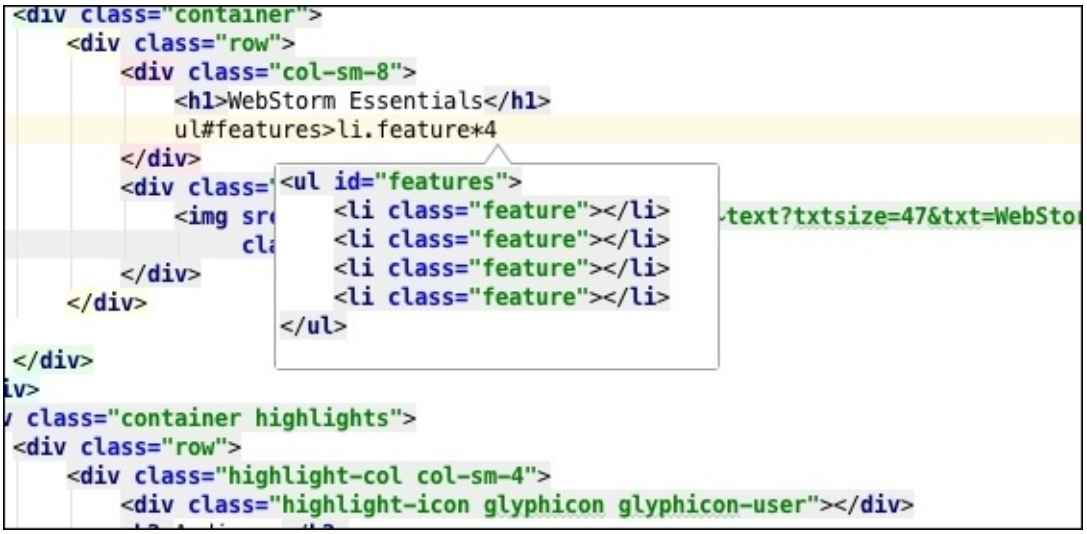
Working with Emmet

Emmet is a set of plugins, which enables the user to use a special syntax that can be transformed to HTML, JSX, or CSS. It is a productivity boost that allows the user to quickly write the code. In WebStorm, the configurations are made in the settings pages found under the **Settings** | **Editor** | **Emmet** page. There are separate configuration pages for **General, HMTL**, **JSX**, and **CSS** settings, as seen in the following screenshot:



Before we start, make sure that you have the Bootstrap example open, and then open the `index.html` and `main.css` files.

While you type your Emmet code, WebStorm will show a preview window if you have selected this in the HTML settings page. It is a good idea to do that in the beginning while you get accustomed to the syntax. The preview will show what your final code will look like.



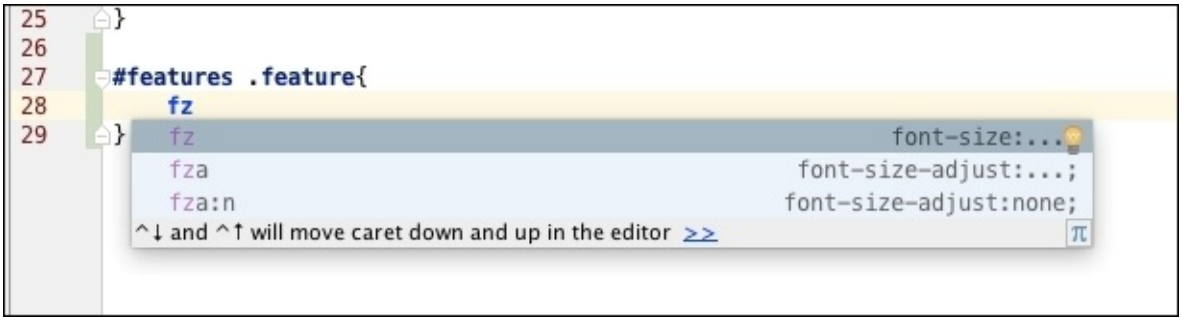
Now you have to insert the sample in the `index.html` file after `<h1>`, so position your caret there, and type `ul#features>li.feature*4`. Then press the *Tab* key. This will expand the code as follows:

```
<ul id="features">
    <li class="feature"></li>
    <li class="feature"></li>
    <li class="feature"></li>
    <li class="feature"></li>
</ul>
```

Now that you have the HTML code, go to the `main.css` file so that you can create the CSS code. Create an empty definition for the element:

```
#features .feature{
}
```

You can now type your Emmet code within this . If you type `fz` and then press *Tab*, this will be expanded to font-size, where we can set the required font size. For the CSS, the preview is always activated.



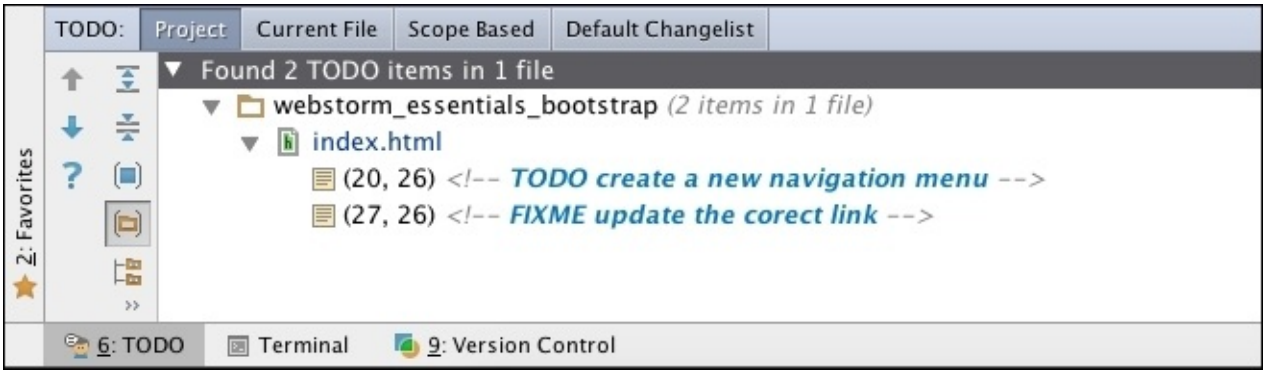
In this section, we have given some simple examples of how Emmet can be used However, if you want to study this subject further, I suggest going through the documentation available at http://docs.emmet.io/.

The TODO facility

WebStorm includes the TODO facility so that you can create a list of tasks for other users or for a future reface. It can be also be used as a notification system that informs the users about issues that require attention such as questions to answer, things to do at a later stage, optimizations, and improvements. It is an indispensable tool when working on large projects where multiple users collaborate.

To use `TODO`, you have to enter special comments in the source code. By default, WebStorm comes with two pre-defined patterns, but you can define as many patterns as you need. The default pattern requires you to create a comment with the `TODO` of the `FIXME` keyword. These special comments are highlighted with special colors that can be defined in the **Preferences** | **Editor** | **Colors and Fonts** | **General** settings page.

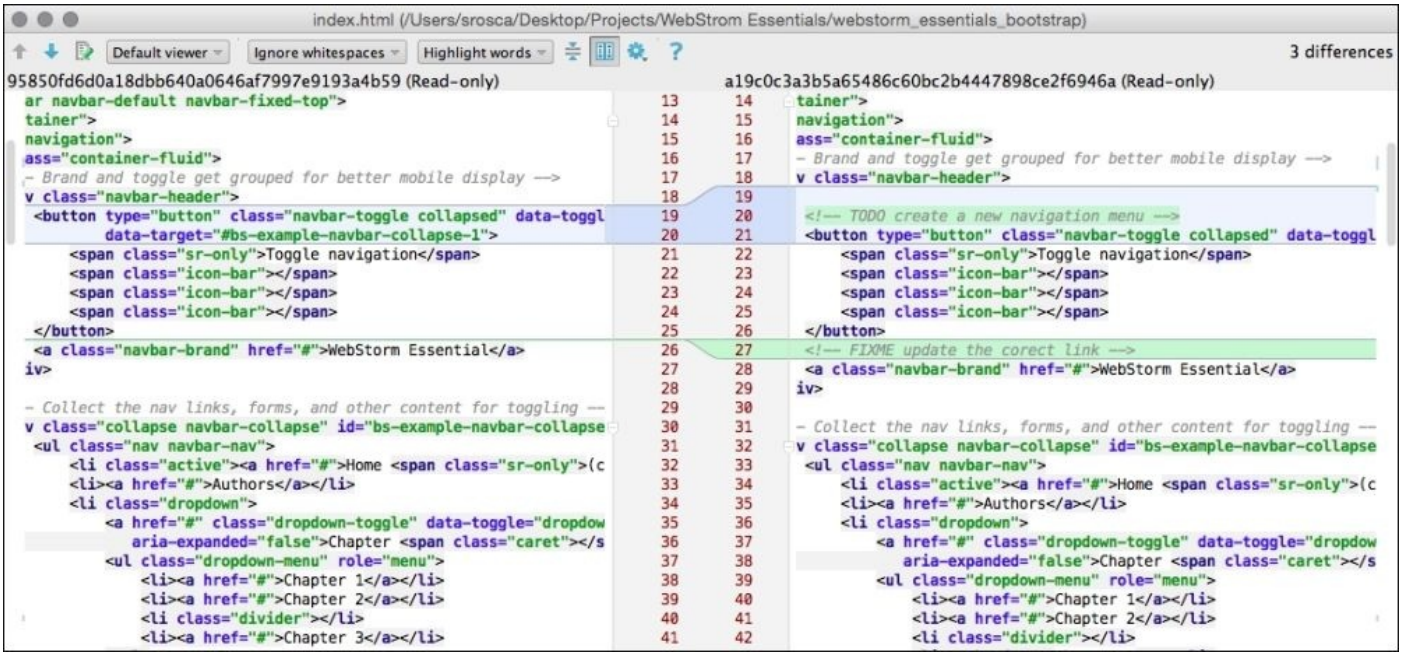
WebStorm also creates a special section were you can see all your TODOs in one place. You can also create filters and custom rules so that you see only the relevant items.



The difference viewer

WebStorm has a special mode that allows us to compare two files or a file at different moments in time (for example, different Git commits). In this mode, we can easily spot the differences between two files or the changes that were made to a file. The differences viewer is also an editor that has code completion, live templates, and so on. Therefore, we can change the code directly in this mode.

The dialog can be accessed by either the **View** | **Compare With** menu or by selecting two files in the **Project** panel. To compare the file with a version from a commit, you need to go to the **Version Control** log, select a file, and then select **Show Diff** from the context menu.

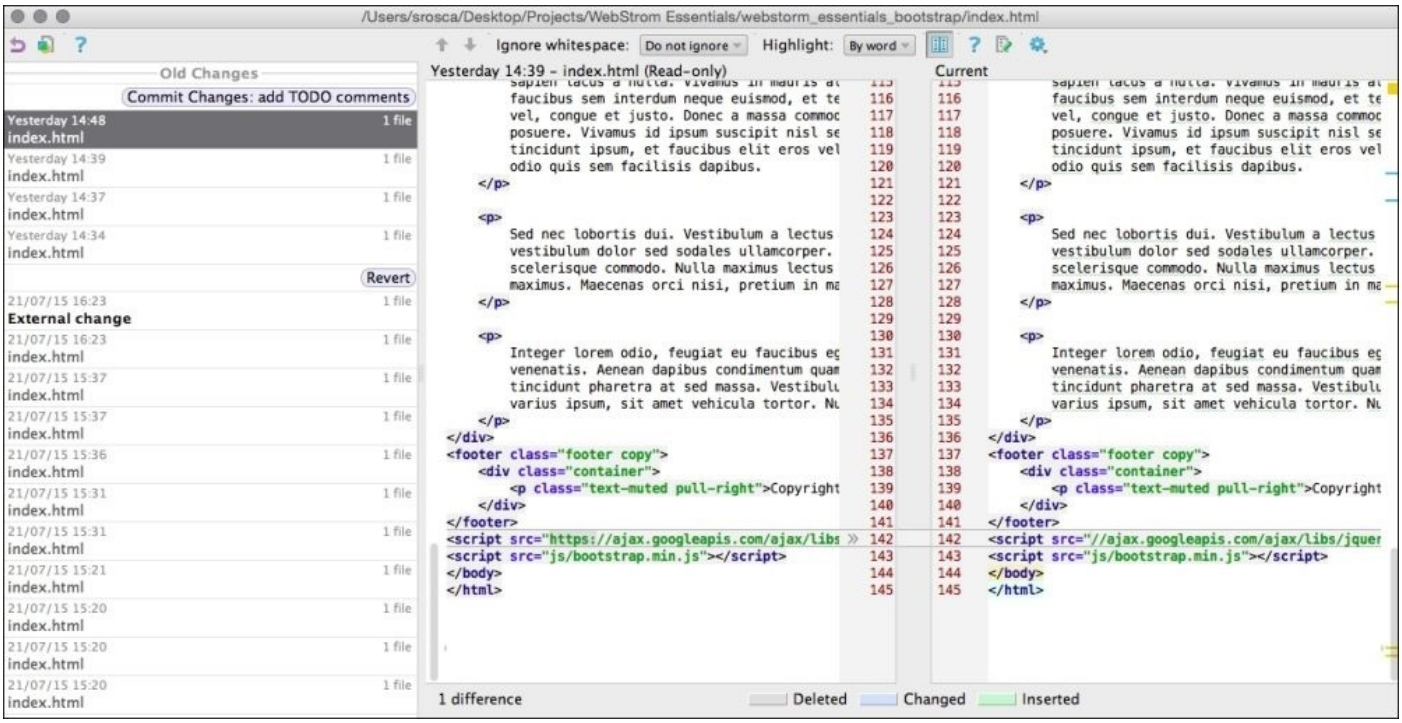


This dialog allows us to quickly see the differences between the files and review any changes.

Tracking Local History

Every file that is modified in WebStorm has the benefit of a powerful local history mode that tracks all the changes made to that file. Local history acts like a local version control system that works in parallel with the other systems that might be attached to the project, and tracks all the changes, even the ones made between commits to the other VCS.

Local History tracks only the changes made to textual files that are smaller than 1 MB. To access the dialog, you need to select **Local History** from the context menu of any opened file.



Summary

In this final chapter, we have gone through some of the features of WebStorm that focus on developer experience and productivity.

We have learned how to use the intelligent editing methods with Live Edit and Emmet, track the tasks that need to be performed in the project, and to view the differences between two files or the changes made to a file over a period of time.

As our journey comes to an end, I hope you have enjoyed learning how WebStorm can help you in your daily tasks. I can say from experience that all the time spent learning this IDE will make you a more confident and productive developer. It is an invaluable tool for development, helping you through your journey from developing simple web pages to more and more complex applications, debugging, and testing of your code.

I would like to end by thanking you for joining me in this journey, and hope you enjoy this new tool in your development toolkit.

Index

A

B

C

D

- differences viewer
  - about / The difference viewer

E

F

- File Watchers
  - about / File Watchers
- Foundation
  - about / Foundation
  - URL / Foundation

G

H

- HTML5 Boilerplate
  - about / HTML5 Boilerplate
  - URL / HTML5 Boilerplate

I

J

- Jasmine
  - about / Jasmine
- JSCS
  - about / JSCS
- JSHint
  - about / JSHint
- JSLint
  - about / JSLint

K

L

M

N

O

- on-the-fly code analysis
  - about / On-the-fly code analysis

P

R

- React
  - about / React
- refactoring facility
  - about / Refactoring facility

S

T

- TODO facility
  - about / The TODO facility

U

V

W

- Wallaby.js
  - about / Wallaby.js
  - URL / Wallaby.js
- Web Starter Kit
  - about / Web Starter Kit
  - URL / Web Starter Kit
- WebStorm
  - installing / Installing WebStorm
  - URL / Installing WebStorm
  - configuring / Configuring WebStorm
  - settings / Settings and preferences
  - preferences / Settings and preferences
- WebStorm, settings
  - about / Settings and preferences
  - themes and colors / Themes and colors
  - keymap / Keymap
  - code style / Code Style
  - languages & frameworks / Languages
  - plugins / Plugins
  - version control / Version Control
  - proxy / Proxy
- WebStorm 10
  - features / What is new in WebStorm 10?
- workspace
  - about / The WebStorm workspace

X