



Community Experience Distilled

Mastering TypeScript

Build enterprise-ready, industrial strength web applications using TypeScript and leading JavaScript frameworks

Nathan Rozentals

[PACKT] open source*
PUBLISHING community experience distilled

www.allitebooks.com

Mastering TypeScript

Build enterprise-ready, industrial strength web applications using TypeScript and leading JavaScript frameworks

Nathan Rozentals

[PACKT] open source 
PUBLISHING community experience distilled
BIRMINGHAM - MUMBAI

Mastering TypeScript

Copyright © 2015 Packt Publishing

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

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

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

First published: April 2015

Production reference: 1170415

Published by Packt Publishing Ltd.
Livery Place
35 Livery Street
Birmingham B3 2PB, UK.

ISBN 978-1-78439-966-5

www.packtpub.com

Credits

Author

Nathan Rozentals

Project Coordinator

Nidhi Joshi

Reviewers

Guy Fergusson

Remo H. Jansen

Andrea Martinelli

Basarat Ali Syed

Proofreaders

Simran Bhogal

Safis Editing

Ameesha Green

Commissioning Editor

Edward Bowkett

Indexer

Rekha Nair

Acquisition Editor

Shaon Basu

Graphics

Abhinash Sahu

Content Development Editor

Kirti Patil

Production Coordinator

Shantanu N. Zagade

Technical Editor

Tanmayee Patil

Cover Work

Shantanu N. Zagade

Copy Editors

Jasmine Nadar

Vikrant Phadke

About the Author

Nathan Rozentals has been writing commercial software for over 24 years. Starting with mainframe COBOL, then moving on to C, followed by C++ and Java, and finally settling on C# and ASP.NET. He has been working with and writing blogs about the TypeScript language, since its release towards the end of 2012. In TypeScript, he found a language through which he could bring all of the object-oriented design patterns and unit testing practices that he had learned over the years, through a variety of languages, to JavaScript.

Nathan currently works in the Health Industry, bringing touch-screen interfaces to medical systems; thereby enabling Bring Your Own Device (BYOD) solutions for clinicians and hospital staff. He is passionate about code quality, unit testing, and Continuous Integration (CI) and has helped many large teams implement CI, across many different software projects, in many different languages.

When he is not coding, Nathan loves windsurfing and playing soccer; he is also an avid Liverpool FC supporter. You can find Nathan's blog at <http://blorkfish.wordpress.com>.

Acknowledgement

I would like to thank my partner, Kathy, for her unconditional love and incredible support over the years. You are simply the best and keep getting better. I am so privileged to have you in my corner. To Matt, thanks for all the laughs, mate; you are a great guy and have so much to give.

To Ayrton and Dayna, I always look forward to hearing from you and finding out about your many exploits. You are constantly in my thoughts and remember that I will always be there for you. To Mum, Rach, Tash, and Tam, thanks for keeping it real guys; there have been times when each of your individual talents has shone through and given me such strength.

To the team at Health (Guy, Dave, Mike, Jeremy, Hardik, Steph, Paul, Marietta, Chris, Wayne, Omar, Hieu, Jes, and yes, even you Kevin), thanks for the many intense debates on all things concerned with software development, architecture, testing, and delivery – we were truly an elite team.

Finally, to Dad, you have always been the voice of reason, the best listener, and by far, the wisest man I have ever known. I am so proud of you, proud all that you have done, in awe of all that you have achieved, and thankful for all that you have taught us.

About the Reviewers

Guy Fergusson wrote his first program on the legendary Texas TI-99/4A in the early eighties. After a successful career in the Royal Australian Air Force during the nineties, he returned to programming in the first decade of the twenty-first century. He has found a new passion in developing responsive web applications using the latest .NET technologies. He was force-fed TypeScript by Nathan Rozentals, and now he wouldn't touch JavaScript without it. He believes there's a future for the Go language and hacks at it in his spare time.

Remo H. Jansen is a web development engineer, open source contributor, entrepreneur, technology lover, gamer, and Internet enthusiast. He is originally from Seville, Spain, but he currently lives in Dublin, Ireland, where he has a full-time job in the funds industry.

Remo has been working on large-scale JavaScript applications for the last few years, from flight booking engines to investment and portfolio management solutions. He loves exploring the possibilities of the Web, learning about new and exciting technologies, and joining knowledge-sharing events such as meets, conferences, and hackathons.

Remo has recently started working as an author on an upcoming book on TypeScript by Packt Publishing.

If you wish to contact him, you will be able to do so at www.remojansen.com.

Andrea Martinelli is a passionate software developer who is currently working on shaman.io, a tool that automatically detects and extracts structured data from the web.

In the past, he has worked on Songr, a music player and aggregator. His interests span across web data extraction, semantic web, code performance and statically typed languages. He is a proficient C# developer and has been interested in TypeScript since its initial announcement. Andrea graduated from the University of Trento in computer science and then studied at the Technical University of Denmark, he is now dedicating more time on the shaman.io project, while moving across Europe.

Basarat Ali Syed (BAS) is a senior developer and the go-to guy for frontend at Picnic Software (<http://picnicsoftware.com/>) in Melbourne, Australia. He studied master of computing at the Australian National University and graduated with high distinction in all courses. He is a familiar face at developer meets and conferences in Australia, and he has been a speaker at events such as ALT.NET, DDD Melbourne, MelbJS, and Node.js meets among others. He is deeply passionate about web technologies. He is a known member of the TypeScript community and works on the DefinitelyTyped team (<https://github.com/DefinitelyTyped>). In his spare time, he enjoys bodybuilding and cycling and maintains a YouTube channel for helping fellow developers (<http://youtube.com/basaratali>).

He is the author of *Beginning Node.js*, *Apress*, and a reviewer of *TypeScript Essentials*, *Packt Publishing*.

You can easily find him on <http://twitter.com/basarat>, <http://github.com/basarat> and simply www.basarat.com.

I would like to thank my family Hasnain, Babar, Baqar, Taskeen and my lovely wife Sana.

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.

Table of Contents

Preface	ix
Chapter 1: TypeScript – Tools and Framework Options	1
What is TypeScript?	3
EcmaScript	3
The benefits of TypeScript	4
Compiling	4
Strong Typing	5
Type definitions for popular JavaScript libraries	6
Encapsulation	8
Public and private accessors	10
TypeScript IDEs	12
Visual Studio 2013	12
Creating a Visual Studio Project	13
Default project settings	14
Debugging in Visual Studio	15
WebStorm	17
Creating a WebStorm project	18
Default files	18
Running the web page in Chrome	22
Debugging in Chrome	23
Brackets	24
Installing Brackets	24
Creating a Brackets project	26
Using Brackets live preview	27
Creating a TypeScript file	28
Compiling our TypeScript	30
Using Grunt	30
Debugging in Chrome	33
Summary	34

Chapter 2: Types, Variables and Function Techniques	35
Basic types	36
JavaScript is not strongly typed	36
TypeScript is strongly typed	37
Type syntax	37
Inferred typing	39
Duck-typing	40
Arrays	41
The any type	42
Explicit casting	42
Enums	43
Const enums	46
Functions	47
Anonymous functions	48
Optional parameters	49
Default parameters	50
The arguments variable	51
Function callbacks	53
Function signatures	54
Function callbacks and scope	56
Function overloads	59
Union types	60
Type guards	61
Type aliases	61
Summary	62
Chapter 3: Interfaces, Classes and Generics	63
Interfaces	64
Classes	65
Class constructors	66
Class functions	67
Interface function definitions	70
Inheritance	71
Interface inheritance	71
Class inheritance	72
Function and constructor overloading with super	73
JavaScript closures	75
The Factory Design Pattern	77
Business requirements	77
What the Factory Design Pattern does	77

The IPerson interface and the Person base class	78
Specialist classes	79
The Factory class	79
Using the Factory class	80
Class modifiers	82
Constructor access modifiers	83
Class property accessors	84
Static functions	85
Static properties	86
Generics	87
Generic syntax	87
Instantiating generic classes	88
Using the type T	90
Constraining the type of T	92
Generic interfaces	94
Creating new objects within generics	95
Runtime type checking	96
Reflection	98
Checking an object for a function	101
Interface checking with generics	102
Summary	105
Chapter 4: Writing and Using Declaration Files	107
Global variables	108
Using JavaScript code blocks in HTML	110
Structured data	111
Writing your own declaration file	112
The module keyword	115
Interfaces	117
Function overrides	120
Rounding out our definition file	122
Module merging	122
Declaration Syntax Reference	123
Function overrides	123
Nested namespaces	124
Classes	124
Class namespaces	125
Class constructor overloads	125
Class properties	126
Class functions	126

Static properties and functions	126
Global functions	127
Function signatures	127
Optional properties	128
Merging functions and modules	128
Summary	129
Chapter 5: Third Party Libraries	131
Downloading definition files	132
Using NuGet	134
Using the Extension Manager	134
Installing declaration files	135
Using the Package Manager Console	136
Installing packages	136
Searching for package names	136
Installing a specific version	136
Using TypeScript Definition Manager	137
Querying for packages	137
Using wildcards	138
Installing definition files	138
Using third party libraries	138
Choosing a JavaScript framework	138
Backbone	139
Using inheritance with Backbone	140
Using interfaces	142
Using generic syntax	143
Using ECMAScript 5	143
Backbone TypeScript compatibility	144
Angular	144
Angular classes and \$scope	146
Angular TypeScript compatibility	148
Inheritance – Angular versus Backbone	149
Angular 2.0	150
ExtJs	150
Creating classes in ExtJs	150
Using type casting	152
ExtJs specific TypeScript compiler	153
Summary	154

Chapter 6: Test Driven Development	155
Test Driven Development	156
Unit, integration and acceptance tests	157
Unit tests	157
Integration tests	157
Acceptance tests	158
Using continuous integration	158
Benefits of continuous integration	159
Selecting a build server	160
Team Foundation Server	160
Jenkins	160
TeamCity	161
Unit testing frameworks	161
Jasmine	161
A simple Jasmine test	162
Jasmine SpecRunner.html file	163
Matchers	165
Test startup and teardown	166
Data-driven tests	167
Using spies	168
Using spies as fakes	170
Asynchronous tests	171
Using the done() function	173
Jasmine fixtures	174
DOM events	175
Jasmine runners	176
Testem	177
Karma	178
Protractor	180
Using Selenium	180
Integration tests	182
Simulating integration tests	182
Detailed test results	185
Logging test results	186
Finding page elements	190
Working with page elements in Jasmine	192
Summary	195

Chapter 7: Modularization	197
CommonJs	198
Setting up Node in Visual Studio	198
Creating a Node module	200
Using a Node module	201
Chaining asynchronous functions	202
Using AMD	206
Backbone	207
Models, collections and views	207
Creating a model	208
The require.config file	210
Fixing Require config errors	213
Using Backbone.Collections	215
Backbone views	219
Using the Text plugin	221
Rendering a collection	223
Creating an application	226
Using jQuery plugins	229
Summary	233
Chapter 8: Object-oriented Programming with TypeScript	235
Program to an interface	236
SOLID principles	236
Single Responsibility	236
Open Closed	237
Liskov Substitution	237
Interface Segregation	237
Dependency Inversion	237
Building a Service Locator	238
The problem space	238
Creating a Service	240
Dependency Resolution	243
Service Location	243
Dependency Injection	243
Service Location versus Dependency Injection	244
A Service Locator	245
Named interfaces	246
Registering classes against named interfaces	247

Using the Service Locator	250
Testability	254
The Domain Events Pattern	255
Problem space	256
Message and Handler Interfaces	257
Multiple Event Handlers	259
Firing an event	262
Registering an Event handler for an Event	264
Displaying error notifications	267
Summary	269
Chapter 9: Let's Get Our Hands Dirty	271
Marionette	271
Bootstrap	272
Board Sales	272
Page layout	272
Installing Bootstrap	273
Using Bootstrap	274
Data structure	277
Data interfaces	278
Integration tests	282
Traversing a collection	284
Finding manufacturer names	286
Finding board types	287
Filtering a Collection	288
Marionette application, regions and layouts	290
Loading the main collection	294
Marionette views	297
The ManufacturerCollectionView class	298
The ManufacturerView class	300
The BoardView class	300
The BoardSizeView class	302
Filtering using the IFilterProvider interface	304
The FilterCollection class	305
Filtering views	308
DOM events in Marionette	310
Triggering a Detail view event	312
Rendering the BoardDetailView	313

Table of Contents

The State Design Pattern	315
Problem space	315
State class diagram	317
Concrete State classes	318
The Mediator class	320
Moving to a new State	323
Implementing the IMediatorFunctions interface	326
Triggering State changes	328
Summary	330
Index	331

Preface

The TypeScript language and compiler has been a huge success story since its release in late 2012. It has quickly carved out a solid footprint in the JavaScript development community, and continues to go from strength to strength. Many large-scale JavaScript projects, including projects by Adobe, Mozilla, and Asana, have made the decision to switch their code base from JavaScript to TypeScript. Recently, the Microsoft and Google teams announced that Angular 2.0 will be developed with TypeScript, thereby merging the AtScript and TypeScript languages into one.

This large-scale industry adoption of TypeScript shows the value of the language, the flexibility of the compiler, and the productivity gains that can be realized with its rich development toolset. On top of this industry support, the ECMAScript 6 standard is getting closer and closer to publication, and TypeScript provides a way to use features of this standard in our applications today.

Writing JavaScript single page applications in TypeScript has been made even more appealing with the large collection of declaration files that have been built by the TypeScript community. These declaration files seamlessly integrate a large range of existing JavaScript frameworks into the TypeScript development environment, bringing with it increased productivity, early error detection, and advanced IntelliSense features.

This book is a guide for both experienced TypeScript developers, as well as those who are just beginning their TypeScript journey. With a focus on Test Driven Development, detailed information on integration with many popular JavaScript libraries, and an in-depth look at TypeScript's features, this book will help you with your exploration of the next step in JavaScript development.

What this book covers

Chapter 1, TypeScript – Tools and Framework Options, sets the scene for beginning TypeScript development, by firstly looking at the various benefits of using TypeScript, and then discussing how to set up a development environment.

Chapter 2, Types, Variables and Function Techniques, introduces the reader to the TypeScript language, starting with basic types and type inferences, and then moving on to discuss variables and functions.

Chapter 3, Interfaces, Classes and Generics, builds on the work from the previous chapter, and introduces the object-oriented concepts of interfaces, classes, and inheritance. It then introduces the reader to the syntax and usage of generics within TypeScript.

Chapter 4, Writing and Using Declaration Files, walks the reader through building a declaration file for an existing body of JavaScript code, and then lists some of the most common syntax used when writing declaration files. This syntax is designed to be a quick reference guide to declaration file syntax, or a cheat sheet.

Chapter 5, Third Party Libraries, shows the reader how to use declaration files from the DefinitelyTyped repository within the development environment. It then moves on to show the reader how to write TypeScript that is compatible with three popular JavaScript frameworks – Backbone, Angular, and ExtJs.

Chapter 6, Test Driven Development, starts with a discussion on what Test Driven Development is, and then guides the reader through the process of creating various types of unit tests using the Jasmine library, including data-driven and asynchronous tests. The chapter finishes with a discussion on integration testing, test reporting, and using continuous integration build servers.

Chapter 7, Modularization, looks at the two types of module generation that the TypeScript compiler uses: CommonJS and AMD. This chapter shows the reader how to build a CommonJS module for use with Node, and then discusses building AMD modules with Require, Backbone, AMD plugins, and jQuery plugins.

Chapter 8, Object-oriented Programming with TypeScript, discusses advanced object-oriented design patterns, including the Service Location Design Pattern, Dependency Injection, and the Domain Events Design Pattern. The reader is taken through the concepts and ideas of each pattern, and then shown how one might implement these patterns using TypeScript.

Chapter 9, Let's Get Our Hands Dirty, builds a single-page application using TypeScript and Marionette from the ground up. This chapter starts with a discussion on page layout and transition, using an HTML-only version of the application. It then moves on to discuss, build and test the underlying data models and Marionette views that will be used within the application. Finally, the State and Mediator Design Pattern is implemented to manage page transitions and graphical elements.

What you need for this book

You will need the TypeScript compiler and an editor of some sort. The TypeScript compiler is available as a Node.js plugin or a Windows executable; therefore, it will run on any operating system. *Chapter 1, TypeScript – Tools and Framework Options* describes the setup of a development environment.

Who this book is for

Whether you are a JavaScript developer wanting to learn TypeScript, or an experienced TypeScript developer wanting to take your skills to the next level, this book is for you. From basic to advanced language constructs, Test Driven Development, and object-oriented techniques, you will learn how to get the most out of the TypeScript language and compiler. This book will show you how to incorporate strong typing, object-orientation, and design best practices into your JavaScript applications.

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: "This GruntFile.js is necessary to setup all of the Grunt tasks."

A block of code is set as follows:

```
class MyClass {
  add(x, y) {
    return x + y;
  }
}
```


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


```
class MyClass {  
  add(x, y) {  
    return x + y;  
  }  
}
```

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

```
tsc app.ts
```

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: "After selecting a **Name** and browsing for a directory, clicking on **OK** will generate a TypeScript project."

 [Warnings or important notes appear in a box like this.]

 [Tips and tricks appear like this.]

Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book—what you liked or 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.

Downloading the color images of this book

We also provide you a PDF file that has color images of the screenshots/diagrams used in this book. The color images will help you better understand the changes in the output. You can download this file from: https://www.packtpub.com/sites/default/files/downloads/B03967_96650S_Graphics.pdf.

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.

1

TypeScript – Tools and Framework Options

JavaScript is a truly ubiquitous language. Just about every website that you visit in the modern world will have some sort of JavaScript component embedded in it, in order to make the site more responsive, more readable, or more attractive to use. Think about the most impressive website that you have visited over the past few months. Was it visually appealing? Did it have some sort of clever presentation? Did it engage you as a user, by giving you a completely new way of discovering car-insurance, or image-sharing, or news articles?

This is the power of JavaScript. JavaScript is the icing on the cake of the internet experience, that makes millions of people around the world go "wow. That's cool". And it generates revenue. Two websites may offer the same product, at the same price, but the one that engages the client – and makes them enjoy the web experience – is the site that will attract the most followers and be the most successful. If this website can also be seamlessly reproduced on desktops, mobiles or tablets, then the target audience – and the target revenue – can be increased exponentially.

On the flip-side, though, JavaScript is also responsible for the annoying side of the Internet. Those annoying advertisements, where you have to wait for 5 seconds before clicking on the **skip** button. Or websites that do not quite work on older browsers, or don't render correctly on tablets and mobile phones. It can be argued that many websites would be better off without JavaScript.

An engaging web experience can also make the difference in corporate web applications. A clunky, difficult to use, and slow web application will turn otherwise keen corporate users completely against your application. Remember that your typical corporate user is comparing their work experience to their daily web experience – of well designed, responsive, intuitive interfaces. After all, they are generally users of the most popular websites out there, and come to expect the same responsiveness at work.

Most of this enhanced user experience comes from the effective use of JavaScript. Asynchronous JavaScript requests allow your web page to render content to the user faster – while waiting for backend processes to do the heavy, time consuming data crunching tasks.

The JavaScript language is not a difficult language to learn, but it does present challenges when writing large, complex programs. Being an interpreted language, JavaScript has no compilation step, and so is executed on the fly. For programmers that are used to writing code in a more formal environment – using compilers, strong typing and well established programming patterns – JavaScript can be a completely foreign environment.

TypeScript bridges this gap. It is a strongly typed, object-oriented, compiled language that allows you as a programmer, to re-use the concepts and ideas of well-established object-oriented languages – in JavaScript. The TypeScript compiler generates JavaScript that adheres to these strongly typed, object-oriented principles – but at the same time is just pure JavaScript. As such, it will run successfully wherever JavaScript can run – in the browser, on the server, or on modern mobile devices.

This chapter is divided into two main sections. The first section is a quick overview of some of the benefits that TypeScript brings to the JavaScript development experience. The second section of this chapter deals with setting up a TypeScript development environment.

If you are an experienced TypeScript programmer, and you already have a development environment set up, then you might want to skip this chapter. If you have never worked with TypeScript before, and have picked up this book because you want to understand what TypeScript can do for you, then read on.

We will cover the following topics in this chapter:

- The benefits of TypeScript
 - Compilation
 - Strong Typing
 - Integration with popular JavaScript libraries

- Encapsulation
- Private and public member variables
- Setting up a development environment
 - Visual Studio
 - WebStorm
 - Brackets and Grunt

What is TypeScript?

TypeScript is both a language and a set of tools to generate JavaScript. It was designed by Anders Hejlsberg at Microsoft (the designer of C#), as an open-source project, to help developers write enterprise scale JavaScript. JavaScript has become widely adopted by programmers around the world – as it can run in any browser on any operating system. With the creation of Node, JavaScript can now also run on the server, desktop or mobile.

TypeScript generates JavaScript – it's as simple as that. Instead of requiring a completely new runtime environment, TypeScript generated JavaScript can re-use all of the existing JavaScript tools, frameworks, and wealth of libraries that are available for JavaScript. The TypeScript language and compiler, however, brings the development of JavaScript closer to a more traditional object-oriented experience.

EcmaScript

JavaScript as a language has been around for a long time, and is also governed by a language feature standard. The language defined in this standard is called ECMAScript, and each browser must deliver functions and features that conform to this standard. The definition of this standard helped the growth of JavaScript and the web in general, and allowed websites to render correctly on many different browsers on many different operating systems. The ECMAScript standard was published in 1999 and is known as ECMA-262, third edition.

With the popularity of the language, and the explosive growth of internet applications, the ECMAScript standard needed to be revised and updated. This process resulted in a draft specification for ECMAScript, called the fourth edition. Unfortunately, this draft suggested a complete overhaul of the language, and was not well received. Eventually, leaders from Yahoo, Google and Microsoft tabled an alternate proposal which they called ECMAScript 3.1. This proposal was numbered 3.1, as it was a smaller feature set of the third edition, and sat "between" edition 3 and 4 of the standard.

This proposal was eventually adopted as the fifth edition of the standard, and was called ECMAScript 5. The ECMAScript fourth edition was never published, but it was decided to merge the best features of both the fourth edition and the 3.1 feature set – into a sixth edition named ECMAScript Harmony.

The TypeScript compiler has a parameter that can be modified to target different versions of the ECMAScript standard. TypeScript currently supports ECMAScript 3, ECMAScript 5 and ECMAScript 6. When the compiler runs over your TypeScript, it will generate compile errors if the code you are attempting to compile is not valid for that particular standard. The team at Microsoft has also committed to follow the ECMAScript standards in any new versions of the TypeScript compiler, so as and when new editions are adopted, the TypeScript language and compiler will follow suit.

An understanding of the finer details of what is included in each release of the ECMAScript standard is outside of the scope of this book, but it is important to know that there are differences. Some browser versions do not support ES5 (IE8 is an example), but most do. When selecting a version of ECMAScript to target for your projects, you will need to consider which browser versions you will be supporting.

The benefits of TypeScript

To give you a flavor of the benefits of TypeScript (and this is by no means the full list), let's take a very quick look at some of the things that TypeScript brings to the table:

- A compilation step
- Strong or static typing
- Type definitions for popular JavaScript libraries
- Encapsulation
- Private and public member variable decorators

Compiling

One of the most frustrating things about JavaScript development is the lack of a compilation step. JavaScript is an interpreted language, and therefore needs to be run in order to test that it is valid. Every JavaScript developer will tell horror stories of hours spent trying to find bugs in their code, only to find that they have missed a stray closing brace `}`, or a simple comma `,` - or even a double quote `"` where there should have been a single quote `'`. Even worse, the real headaches arrive when you misspell a property name, or unwittingly re-assign a global variable.

TypeScript will compile your code, and generate compilation errors where it finds these sort of syntax errors. This is obviously very useful, and can help to highlight errors before the JavaScript is run. In large projects, programmers will often need to do large code merges – and with today's tools doing automatic merges – it is surprising how often the compiler will pick up these types of errors.

While tools to do this sort of syntax checking – like JSLint – have been around for years, it is obviously beneficial to have these tools integrated into your IDE. Using TypeScript in a continuous integration environment will also fail a build completely when compilation errors are found – further protecting your programmers against these types of bugs.

Strong Typing

JavaScript is not strongly typed. It is a language that is very dynamic, as it allows objects to change their properties and behavior on the fly. As an example of this, consider the following code:

```
var test = "this is a string";
test = 1;
test = function(a, b) {
    return a + b;
}
```

On the first line of this code snippet, the variable `test` is bound to a string. It is then assigned a number, and finally is redefined to be a function that expects two parameters. Traditional object oriented languages, however, will not allow the type of a variable to change – hence they are called strongly typed languages.

While all of the preceding code is valid JavaScript - and could be justified - it is quite easy to see how this could cause runtime errors during execution. Imagine that you were responsible for writing a library function to add two numbers, and then another developer inadvertently re-assigned your function to instead subtract these numbers.

These types of errors may be easy to spot in a few lines of code, but it becomes increasingly difficult to find and fix these as your code base, and your development team grows.

Another feature of strong typing is that the IDE you are working in can understand what type of variable you are working with, and can bring better autocomplete or Intellisense options to the fore.

TypeScript's "syntactic sugar"

TypeScript introduces a very simple syntax to check the type of an object at compile time. This syntax has been referred to as "syntactic sugar", or more formally, type annotations. Consider the following TypeScript code:

```
var test: string = "this is a string";
test = 1;
test = function(a, b) { return a + b; }
```

Note on the first line of this code snippet, we have introduced a colon `:` and a `string` keyword between our variable and its assignment. This type annotation syntax means that we are setting the type of our variable to be of type `string`, and that any code that does not use it as a string will generate a compile error. Running the preceding code through the TypeScript compiler will generate two errors:

```
error TS2011: Build: Cannot convert 'number' to 'string'.
error TS2011: Build: Cannot convert '(a: any, b: any) => any' to
'string'.
```

The first error is fairly obvious. We have specified that the variable `test` is a `string`, and therefore attempting to assign a number to it will generate a compile error. The second error is similar to the first, and is in essence saying that we cannot assign a function to a string.

In this way, the TypeScript compiler introduces strong, or static typing to your JavaScript code, giving you all of the benefits of a strongly typed language. TypeScript is therefore described as a "superset" of JavaScript. We will explore typing in more detail in the next chapter.

Type definitions for popular JavaScript libraries

As we have seen, TypeScript has the ability to "annotate" JavaScript, and bring strong typing to the JavaScript development experience. But how do we strongly type existing JavaScript libraries? The answer is surprisingly simple: by creating a definition file. TypeScript uses files with a `.d.ts` extension as a sort of "header" file, similar to languages such as C++, to superimpose strongly typing on existing JavaScript libraries. These definition files hold information that describes each available function and variable of the library, along with their associated type annotations.

Let's take a quick look at what a definition would look like. As an example, consider a function from the popular Jasmine unit testing framework called `describe`:

```
var describe = function(description, specDefinitions) {
  return jasmine.getEnv().describe(description, specDefinitions);
};
```

This function has two parameters, `description` and `specDefinitions`. Just reading this JavaScript, however, does not tell us what sort of parameters these are meant to be. Is the `specDefinitions` argument a string, or an array of strings, a function or something else? In order to figure this out, we would need to have a look through the Jasmine documentation found at <http://jasmine.github.io/2.0/introduction.html>. This documentation provides us with a helpful sample of how to use this function:

```
describe("A suite", function () {
  it("contains spec with an expectation", function () {
    expect(true).toBe(true);
  });
});
```

From the documentation, then, we can easily see that the first parameter is a string, and the second parameter is a function. There is nothing in the JavaScript language, however, that forces us to conform to this API. As mentioned before, we could easily call this function with two numbers – or inadvertently switch the parameters around, sending a function first, and a string second. We will obviously start getting runtime errors if we do this, but TypeScript – using a definition file – can generate compile time errors before we even attempt to run this code.

Let's take a look at a piece of the `jasmine.d.ts` definition file:

```
declare function describe(
  description: string, specDefinitions: () => void
): void;
```

This is the TypeScript definition for the `describe` function. Firstly, `declare function describe` tells us that we can use a function called `describe`, but that the implementation of this function will be provided at runtime.

Clearly, the `description` parameter is strongly typed to be of type `string`, and the `specDefinitions` parameter is strongly typed to be a function that returns `void`. TypeScript uses the double braces `()` syntax to declare functions, and the fat arrow syntax to show the return type of the function. So `() => void` is a function that does not return anything. Finally, the `describe` function itself will return `void`.

If our code were to try and pass in a function as the first parameter, and a string as the second parameter (clearly breaking the definition of this function) as shown in the following example:

```
describe(() => { /* function body */}, "description");
```

The TypeScript compiler will immediately generate the following errors:

```
error TS2082: Build: Supplied parameters do not match any signature of call target: Could not apply type "string" to argument 1 which is of type () => void
```

This error is telling us that we are attempting to call the `describe` function with invalid parameters. We will look at definition files in more detail in later chapters, but this example clearly shows that TypeScript will generate errors if we attempt to use external JavaScript libraries incorrectly.

Definitely Typed

Soon after TypeScript was released, Boris Yankov started a GitHub repository to house definition files, at DefinitelyTyped (<https://github.com/borisyankov/DefinitelyTyped>). This repository has now become the first port of call for integrating external libraries into TypeScript, and currently holds definitions for over 500 JavaScript Libraries.

Encapsulation

One of the fundamental principles of object-oriented programming is encapsulation: The ability to define data, as well as a set of functions that can operate on that data, into a single component. Most programming languages have the concept of a class for this purpose – providing a way to define a template for data and related functions.

Let's first take a look at a simple TypeScript class definition:

```
class MyClass {
  add(x, y) {
    return x + y;
  }
}

var classInstance = new MyClass();
console.log(classInstance.add(1, 2));
```

This code is pretty simple to read and understand. We have created a `class`, named `MyClass`, with a single function named `add`. To use this class, we simply create an instance of it, and call the `add` function with two arguments.

JavaScript, unfortunately, does not have a `class` keyword, but instead uses functions to reproduce the functionality of classes. Encapsulation through classes is accomplished by either using the prototype pattern, or by using the closure pattern. Understanding prototypes and the closure pattern, and using them correctly, is considered a fundamental skill when writing enterprise-scale JavaScript.

A closure is essentially a function that refers to independent variables. This means that variables defined within a closure function 'remember' the environment in which they were created. This provides JavaScript with a way to define local variables, and provide encapsulation. Writing the `MyClass` definition in the preceding code, using a closure in JavaScript would look something like this:

```
var MyClass = (function () {
    // the self-invoking function is the
    // environment that will be remembered
    // by the closure
    function MyClass() {
        // MyClass is the inner function,
        // the closure
        MyClass.prototype.add = function (x, y) {
            return x + y;
        };
        return MyClass;
    }()
}) ();
var classInstance = new MyClass();
console.log("result : " + classInstance.add(1, 2));
```

We start with a variable called `MyClass`, and assign it to a function that is executed immediately - note the `})();` syntax near the bottom of the code snippet. This syntax is a common way to write JavaScript in order to avoid leaking variables into the global namespace. We then define a new function named `MyClass`, and return this new function to the outer calling function. We then use the `prototype` keyword to inject a new function into the `MyClass` definition. This function is named `add` and takes two parameters, returning their sum.

The last two lines of the code show how to use this closure in JavaScript. Create an instance of the closure type, and then execute the `add` function. Running this in the browser will log **result: 3** to the console, as expected.




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.

Looking at the JavaScript code versus the TypeScript code, we can easily see how simple TypeScript looks, compared to the equivalent JavaScript. Remember how we mentioned that JavaScript programmers can easily misplace a brace {, or a bracket (? Take a look at the last line in the closure definition: }) () ; Getting one of these brackets or braces wrong can take hours of debugging to find.

TypeScript classes generate closures

The JavaScript closure as shown in the preceding code snippet, is actually the output of the TypeScript class definition. So TypeScript actually generates closures for you.

 Adding the concept of classes to the JavaScript language has been talked about for years, and is currently a part of the ECMAScript sixth Edition (Harmony) standard – but this is still a work in progress. Microsoft has committed to follow the ECMAScript standard in the TypeScript compiler, as and when these standards are published.

Public and private accessors

A further object-oriented principle that is used in encapsulation is the concept of data hiding – the ability to have public and private variables. Private variables are meant to be hidden to the user of a particular class – as these variables should only be used by the class itself. Inadvertently exposing these variables outside of a class can easily cause runtime errors.

Unfortunately, JavaScript does not have a native way of declaring variables private. While this functionality can be emulated using closures, a lot of JavaScript programmers simply use the underscore character _ to denote a private variable. At runtime though, if you know the name of a private variable – you can easily assign a value to it. Consider the following JavaScript code:

```
var MyClass = (function() {
  function MyClass() {
    this._count = 0;
  }
  MyClass.prototype.countUp = function() {
    this._count ++;
  }
  MyClass.prototype.getCountUp = function() {
    return this._count;
  }
})
```

```
        return MyClass;
    }());

    var test = new MyClass();
    test._count = 17;
    console.log("countUp : " + test.getCountUp());
```

The `MyClass` variable is actually a closure – with a constructor function, a `countUp` function and a `getCountUp` function. The variable `_count` is supposed to be a private member variable, one that is used only within the scope of the closure. Using the underscore naming convention gives the user of this class some indication that the variable is private, but JavaScript will still allow you to manipulate the variable `_count`. Take a look at the second last line of the code snippet. We are explicitly setting the value of the supposed private variable `_count` to 17 – which is allowed by JavaScript, but not desired by the original creator of the class. The output of this code would be **countUp: 17**.

TypeScript, however, introduces the `public` and `private` keywords that can be used on class member variables. Trying to access a class member variable that has been marked as `private` will generate a compile time error. As an example of this, the JavaScript code above can be written in TypeScript as follows:

```
class MyClass {
    private _count: number;
    constructor() {
        this._count = 0;
    }
    countUp() {
        this._count++;
    }
    getCount() {
        return this._count;
    }
}

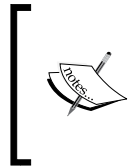
var classInstance = new MyClass();
console.log(classInstance._count);
```

On the second line of our code snippet, we have declared a private member variable named `_count`. Again, we have a constructor, a `countUp` and a `getCount` function. If we compile this TypeScript code, the compiler will generate an error:

```
error TS2107: Build: 'MyClass._count' is inaccessible.
```

This error is generated because we are trying to access the private variable `_count` in the last line of the code.

The TypeScript compiler, therefore, is helping us to adhere to public and private accessors – by generating a compile error when we inadvertently break this rule.



Remember, though, that these accessors are a compile-time feature only, and will not affect the generated JavaScript. You will need to bear this in mind if you are writing JavaScript libraries that will be consumed by third parties. The TypeScript compiler will also still generate the JavaScript output file, even if there are compile errors.

TypeScript IDEs

The purpose of this section is to get you up and running with a TypeScript environment so that you can edit, compile, run and debug your TypeScript code. TypeScript has been released as open-source, and includes both a Windows variant, and a Node variant. This means that the compiler will run on Windows, Linux, OS X, and any other operating system that supports Node.

On Windows environments, we can either install Visual Studio – which will register the `tsc.exe` (TypeScript Compiler) in our `C:\Program Files` directory, or we can use Node. On Linux and OS X environments, we will need to use Node. Either way, firing up a command prompt and typing `tsc -v` should display the current version of the compiler that we are using. Which at the time of writing, is version 1.4.2.0.

In this section, we will be looking at the following IDEs:

- Visual Studio 2013
- WebStorm
- Brackets

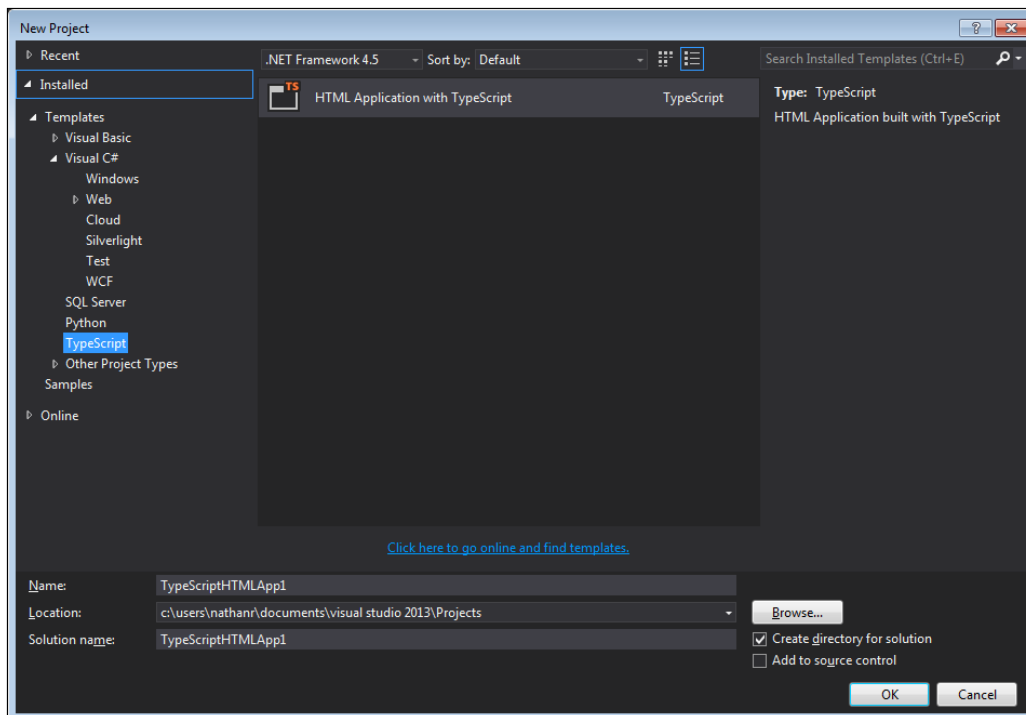
Visual Studio 2013

First up, let's look at Microsoft's Visual Studio 2013. This is Microsoft's primary IDE, and comes in a variety of pricing combinations. At the top end is Ultimate, then Premium, then Professional, and finally Express. Ultimate, Premium and Professional all require paid licenses which range (at the time of writing) from \$13,000 through to \$1,199. The good news is that Microsoft has recently announced a Community Edition, which can be used in non-enterprise environments for both free and non-paid products. The TypeScript compiler is included in all of these editions.

Visual Studio can be downloaded as either a web-installer, or an .ISO CD image. Note that the web installer will require an internet connection during installation, as it downloads the required packages during the installation step. Visual Studio will also require Internet Explorer 10 or later, but will prompt you during installation, if you have not upgraded your browser as yet. If you are using the .ISO installer, just bear in mind that you may be required to download and install additional operating system patches if you have not updated your system with Windows Update in a while.

Creating a Visual Studio Project

Once Visual Studio is installed, fire it up and create a new project (**File | New Project**). Under the **Templates** section on the left hand side, you will see a TypeScript option. When this option is selected, you will be able to use a project template named **Html Application with TypeScript**. Enter a name and location for your project, and then click **OK** to generate a TypeScript project:



Visual Studio – selecting the TypeScript project type



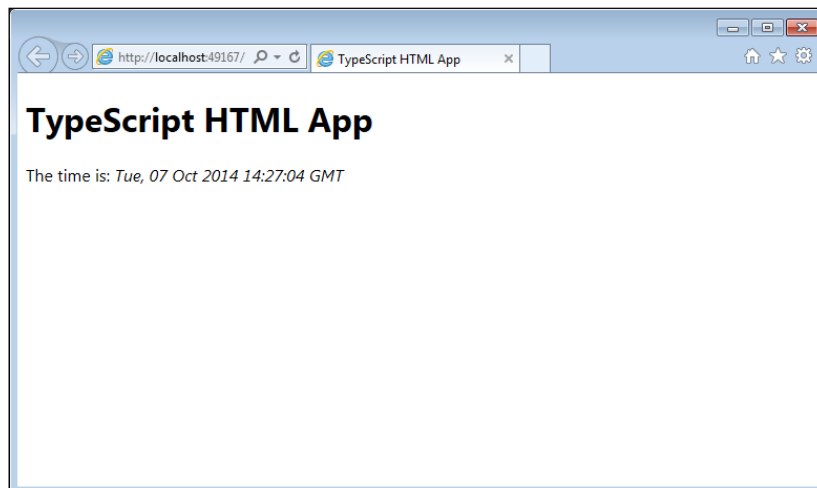
This is not the only project template that works with TypeScript. Any of the ASP.NET project types support TypeScript out of the box. If you are planning to use the Web API to provide RESTful data controllers, then you may consider creating an MVC Web Application from the start. Then, by simply including a TypeScript file, and specifying a `.ts` file extension within the project, Visual Studio will automatically start compiling your TypeScript files as part of the new project.

Default project settings

Once a new TypeScript project is created, notice that the project template generates a few files for us automatically:

- `app.css`
- `app.ts`
- `index.html`
- `web.config`

If we were to compile and then run this project now, we would have a complete, running TypeScript application right off the bat:



Visual Studio index.html running in Internet Explorer

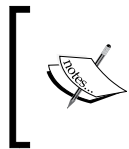
Let's take a quick look at the generated `index.html` file and what it contains:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>TypeScript HTML App</title>
  <link rel="stylesheet" href="app.css" type="text/css" />
  <script src="app.js"></script>
</head>
<body>
  <h1>TypeScript HTML App</h1>

  <div id="content"></div>
</body>
</html>
```

This is a very simple HTML file, which includes the `app.css` style sheet, as well as a JavaScript file named `app.js`. This `app.js` file is the JavaScript file that is generated from `app.ts` TypeScript file, when the project is compiled.



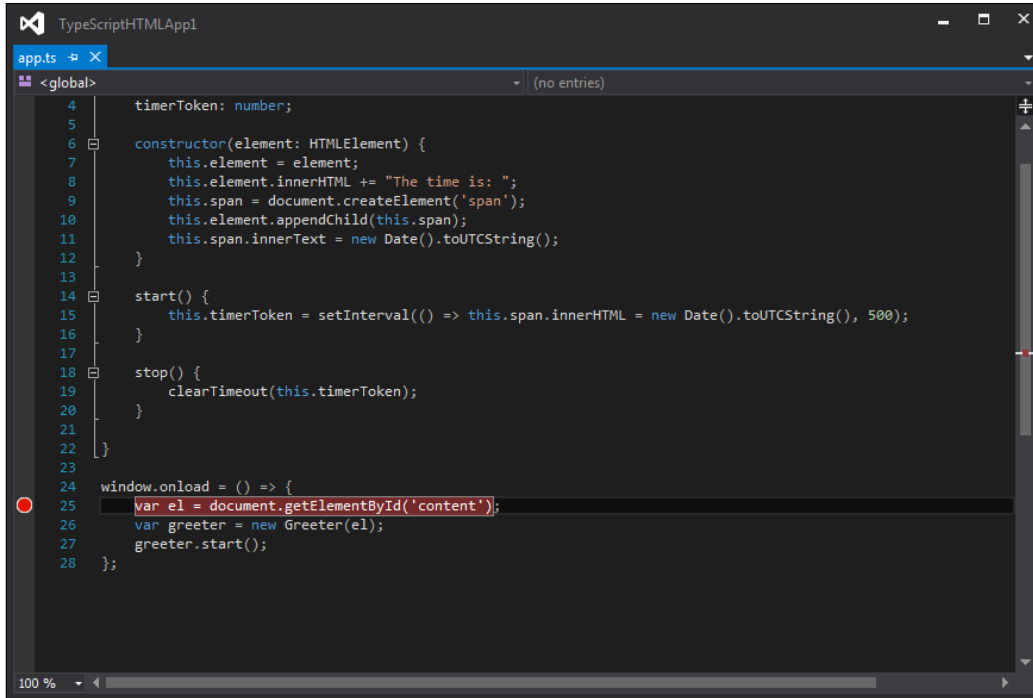
The `app.js` file is not included in the **Solution Explorer** – only the `app.ts` TypeScript file is included. This is by design. If you wish to see the generated JavaScript file, simply click on the **Show All Files** button in the **Solution Explorer** toolbar.

Debugging in Visual Studio

One of the best features of Visual Studio is that it is truly an integrated environment. Debugging TypeScript in Visual Studio is exactly the same as debugging C# – or any other language in Visual Studio – and includes the usual **Immediate**, **Locals**, **Watch** and **Call stack** windows.


To debug TypeScript in Visual Studio, simply put a breakpoint on the line you wish to break on in your TypeScript file (Move your mouse into the breakpoint area next to the source code line, and click). In the image below, we have placed a breakpoint within the `window.onload` function.

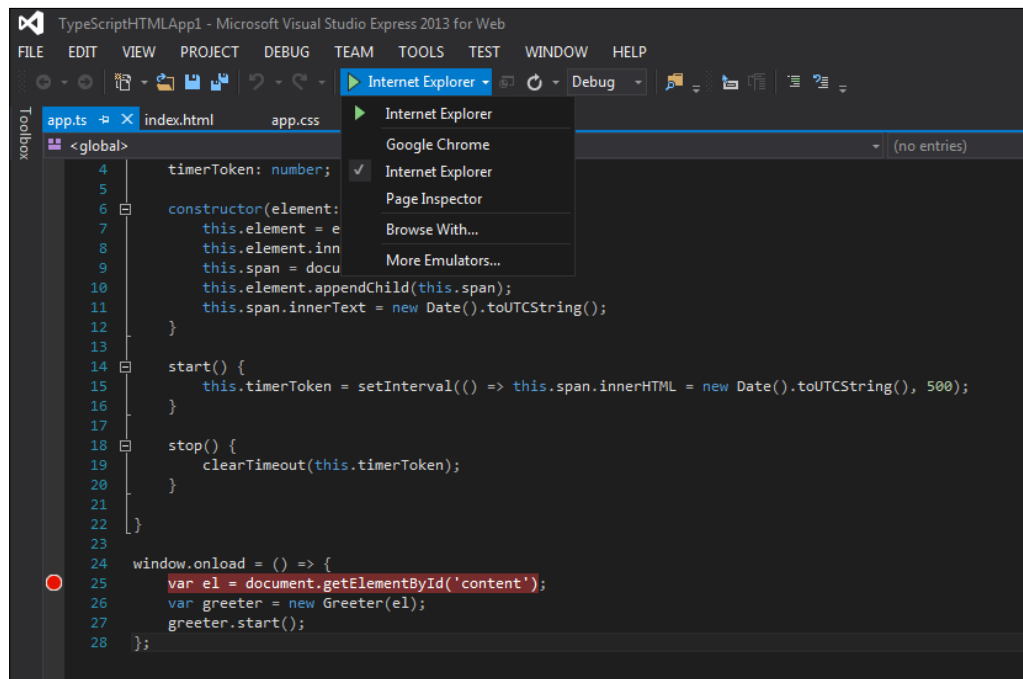
To start debugging, simply hit F5.



Visual Studio TypeScript editor with a breakpoint set in the code

When the source code line is highlighted in yellow, simply hover your mouse over any of the variables in your source, or use the **Immediate**, **Watch**, **Locals** or **Call stack** windows.

 Visual Studio only supports debugging in Internet Explorer. If you have multiple browsers installed on your machine, make sure that you select Internet Explorer in your **Debug** toolbar, as shown in the following screenshot:



Visual Studio debug toolbar showing browser options

WebStorm

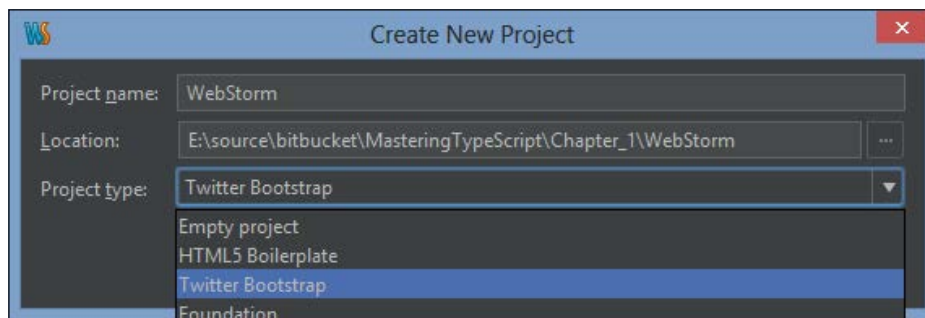
WebStorm is a popular IDE by JetBrains (<http://www.jetbrains.com/webstorm/>), and will run on Windows, Mac OS X and Linux. Prices range from \$49 for a single developer to \$99 for a commercial license. JetBrains also offers a 30 day trial version.

WebStorm has a couple of great features, including live-edit and code suggestions, or Intellisense. The live-edit feature allows you to keep a browser window open, which WebStorm will automatically update based on changes to CSS, HTML and JavaScript as you type it. Code suggestions – which are also available with another popular JetBrains product named ReSharper – will highlight code that you have written, and suggest better ways of implementing it. WebStorm also has a large number of project templates. These templates will automatically download and include the relevant JavaScript or CSS files needed by the template, such as Twitter Bootstrap, or HTML5 boilerplate.

Setting up WebStorm is as simple as downloading the package from the website, and running the installer.

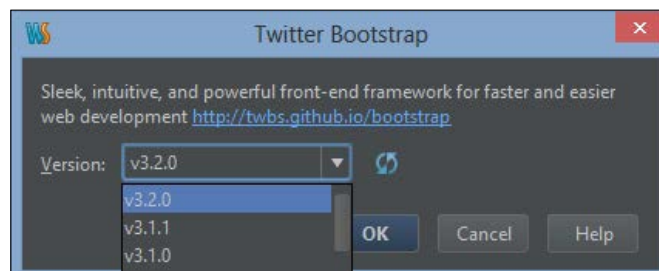
Creating a WebStorm project

To create a new WebStorm project, simply fire it up, and hit **File | New Project**. Select a **Name**, **Location** and **Project type**. For this project, we have chosen `Twitter Bootstrap` as the project type, as shown in the following screenshot:



WebStorm Create New Project dialog box

WebStorm will then ask you to select the version of Twitter Bootstrap that you intend developing for. In this example, we have chosen version `v3.2.0`.



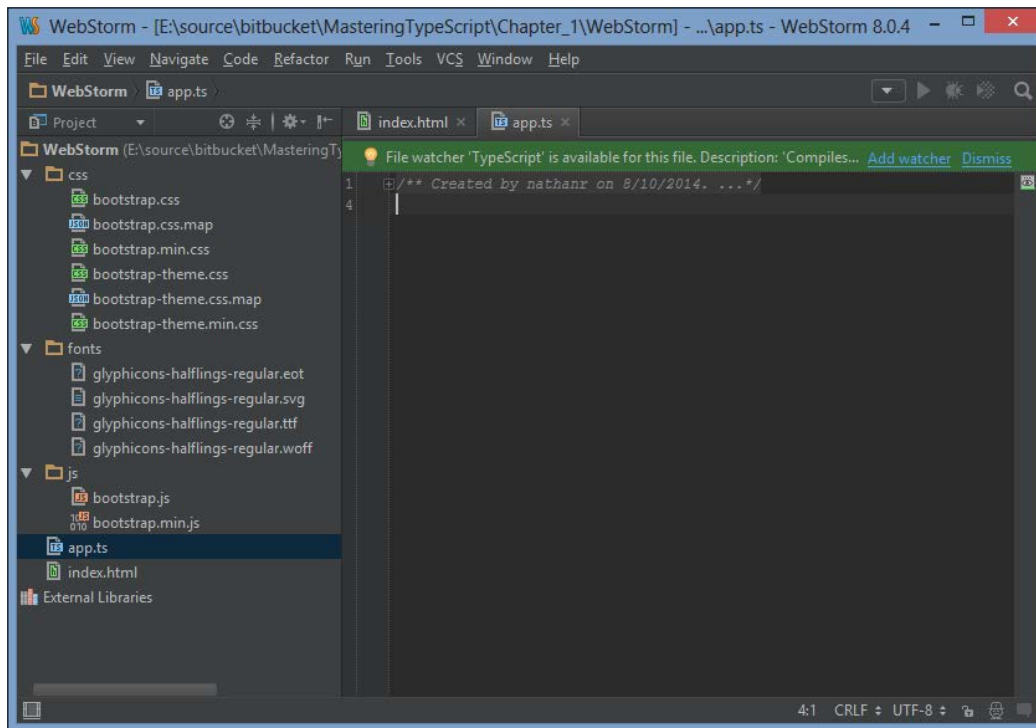
WebStorm Select Twitter Bootstrap version dialog box

Default files

WebStorm has conveniently created a `css`, `fonts` and `js` directory as part of the new project – and downloaded and included the relevant CSS, font files and JavaScript files for us, in order to start building a new Bootstrap based website. Note that it has not created an `index.html` file for us, nor has it created any TypeScript files – as Visual Studio did. After a while working with TypeScript, most developers will delete these generic files anyway. So let's create an `index.html` file.

Simply click on **File | New**, select HTML file, enter `index` as a name, and click **OK**.

Next, let's create a TypeScript file in a similar manner. We will call this file `app` (or `app.ts`), to be the same as in the Visual Studio default project example. As we click inside the new `app.ts` file, WebStorm will pop up a green bar at the top of our edit window, with a suggestion reading **File watcher 'TypeScript' is available for this file**, as shown in the following screenshot:



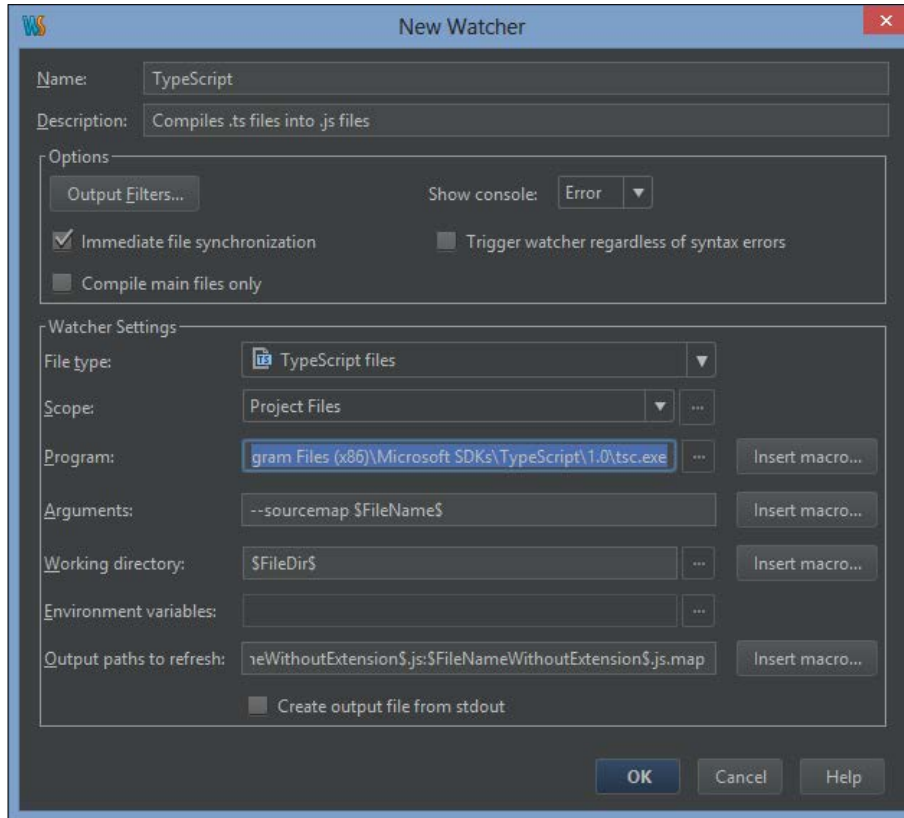
WebStorm editing a TypeScript file for the first time showing the File Watcher bar

A WebStorm "file watcher" is a background process that will execute as soon as you have saved the file. This is equivalent to Visual Studio's **Compile on save** TypeScript option. As WebStorm suggests, now would be a good time to activate this file watcher for TypeScript. Click on the **Add watcher** link in the green bar, and fill in the details on the next screen.

We can leave the defaults on the next screen as they are for the time being, except for the **Program** setting:

If you are running on Windows, and already have Visual Studio installed, then this should be set to the full path of the `tsc.exe` executable, i.e. `C:\Program Files (x86)\Microsoft SDKs\TypeScript\1.0\tsc.exe`, as shown in the following screenshot:

If you are running on a non-windows box, or have installed TypeScript via Node, then this would just be set to `tsc`, with no path.



WebStorm new file watcher options screen

Now that we have a file watcher created for our TypeScript files, let's create a simple TypeScript class, which will modify the `innerText` of an HTML `div`. While you are typing, you will notice WebStorm's autocompletion or Intellisense feature helping you with available keywords, parameters, naming conventions and a host of other language specific information. This is one of the most powerful features of WebStorm, and is similar to the enhanced Intellisense seen in JetBrains' Resharper tool for Visual Studio. Go ahead and type the following TypeScript code, during which you will get a good feeling of WebStorm's available autocompletion.

```
class MyClass {
  public render(divId: string, text: string) {
    var el: HTMLElement = document.getElementById(divId);
    el.innerText = text;
  }
}

window.onload = () => {
  var myClass = new MyClass();
  myClass.render("content", "Hello World");
}
```

We start off with the `MyClass` class definition, which simply has a function called `render`. This `render` function takes a DOM element name, and a text string as parameters. It then simply finds the DOM element, and sets the `innerText` property. Note the use of strong typing on the variable `el` - we have explicitly typed this to be of the `HTMLElement` type.

We are also assigning a function to the `window.onload` event, which will execute once the page has been loaded, similar to the Visual Studio sample. Within this function, we are simply creating an instance of `MyClass`, and calling the `render` function with two string arguments.

If you have any errors in your TypeScript file, these will automatically show up in the output window, giving you instant feedback while you type. With this TypeScript file created, we can now include it in our `index.html` file, and try some debugging.

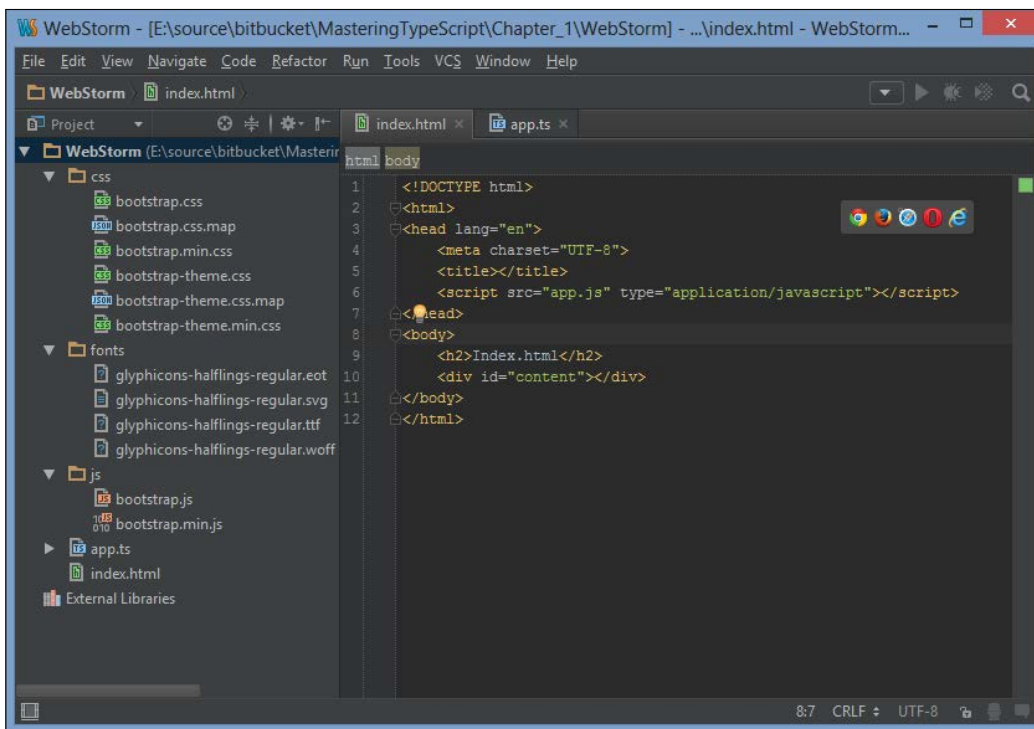
Open the `index.html` file, and add a script tag to include the `app.js` JavaScript file, along with a `div` with an `id` of `"content"`. Just as we saw with TypeScript editing, you will find that WebStorm has powerful Intellisense features when editing HTML as well.

```
<!DOCTYPE html>
<html>
<head lang="en">
  <meta charset="UTF-8">
  <title></title>
  <script src="app.js" type="application/javascript"></script>
</head>
<body>
  <h2>Index.html</h2>
  <div id="content"></div>
</body>
</html>
```

There are a couple of points to note in the preceding code. We are including a script tag for the `app.js` JavaScript file, as this is the output file that the TypeScript compiler will generate. We have also created an HTML `<div>` with an id of `content` that the instance of the `MyClass` class will use to render our text.

Running the web page in Chrome

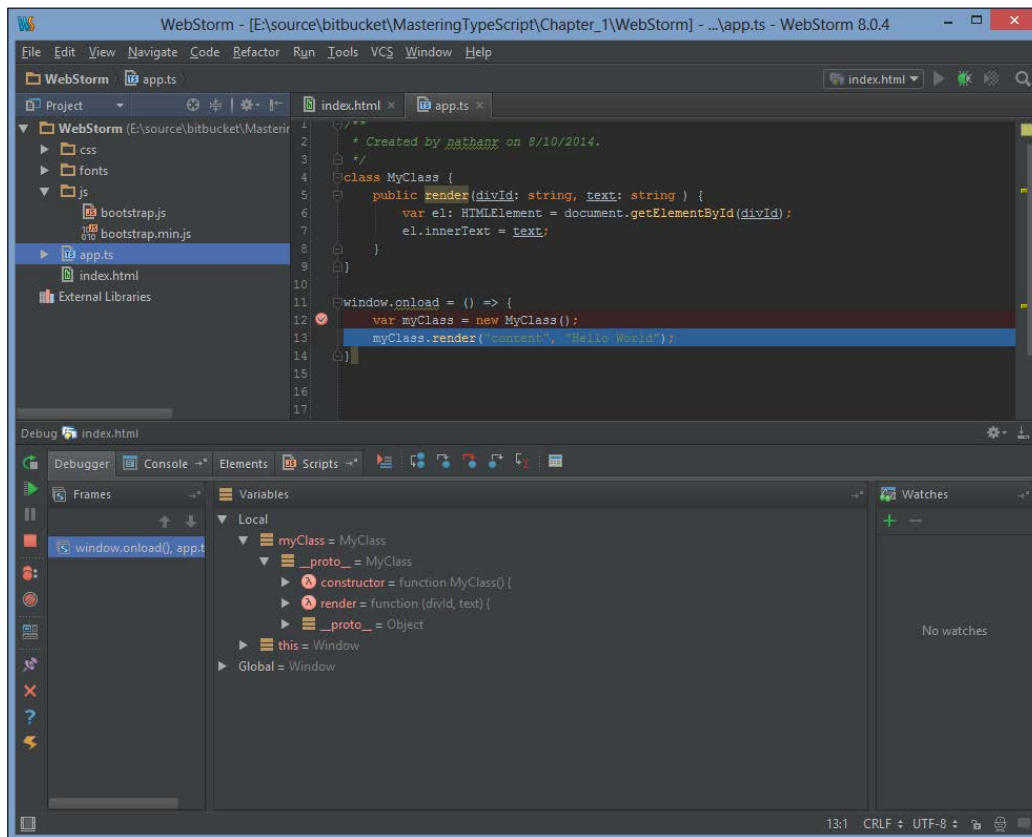
When viewing or editing HTML files in WebStorm, you will notice a small set of browser icons popping up on the top right corner of the editing window. Clicking on any one of the icons will launch your current HTML page using the selected browser.



WebStorm editing an HTML file showing the popup browser launching icons

Debugging in Chrome

As we saw in Visual Studio, debugging in WebStorm is simply a matter of marking a breakpoint, and then hitting *Alt + F5*. WebStorm uses a Chrome Plugin to enable debugging in Chrome. If you do not have this plugin installed, WebStorm will prompt you the first time you start debugging, to download and enable the JetBrains IDE Support Chrome Plugin. With this plugin enabled, WebStorm has a very powerful set of tools to inspect JavaScript code, add watchers, view the console and many more, right inside the IDE.



WebStorm debugging session showing debugger panels

Brackets

The last IDE that we will look at in this chapter is not really an IDE for TypeScript, it is more of an IDE for web designers that has TypeScript editing capability. Brackets is an open-source code editor, and is really good at helping design and style webpages. Similar to WebStorm, it has a live editing mode where you can see changes to HTML or CSS on the running web page as you type. In our development teams, Brackets has become a very popular editor for rapid prototyping of HTML web pages and CSS styling.

There are a couple of reasons to include Brackets in this chapter. Firstly, it is completely open-source and therefore completely free – and it runs on Windows, Linux and Mac OS X. Secondly, using a Brackets environment shows what a bare-bones TypeScript environment would look like, with just a text editor and the command line. Lastly, Brackets shows that the syntax highlighting and code-completion capability of open-source projects can be just as good – if not faster than commercial IDEs.

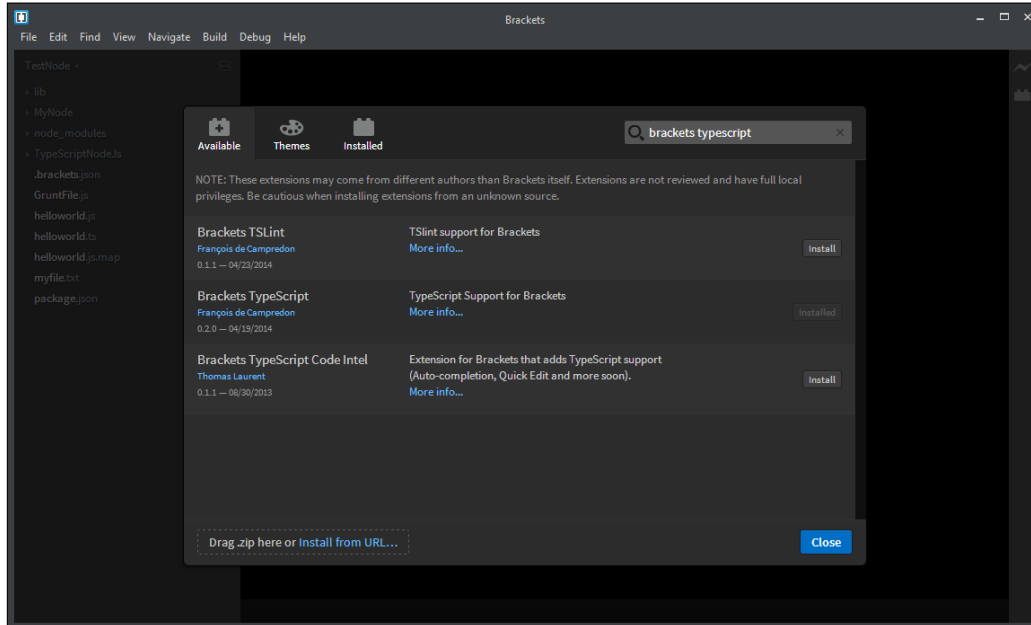
Installing Brackets

Brackets can be downloaded with the preferred installers from <http://brackets.io>. Once installed, we will need to install some extensions. Brackets has a really slick and simple extension manager, which is easy to use, and which allows us to easily find and install available extensions. Any time an update to either Brackets, or one of your installed extensions is available, Brackets will automatically notify you.

To install an extension, fire up Brackets, and either click on **File | Extension Manager**, or click on the lego-block icon on the right-hand side vertical sidebar.

To start with, we will need to install the TypeScript extension. In the search bar, type `brackets typescript`, and install the **Brackets TypeScript** extension from **Francois de Campredon**.

As can be seen from the following screenshot, each extension has a **More info...** link – which will take you to the extension home page.



Brackets Extension manager interface

As well as the **Brackets TypeScript** extension, another useful extension is **Code Folding** by **Patrick Oladimeji**. This will allow you to collapse or expand sections of code in any file that you are editing.

Another great time-saver is **Emmet** by **Sergey Chikujonok**. Emmet (previously known as Zen Coding) uses a CSS-like short-hand, instead of traditional code snippets, to generate HTML. In this section, we will quickly show how Emmet can be used to generate HTML, just as a teaser. So go ahead and install the Emmet extension.

Creating a Brackets project

Brackets does not have the concept of a project per se, but instead just works off a root folder. Create a directory on your filesystem, and then open that folder in Brackets: **File | Open Folder**.

Let's now create a simple HTML page using Brackets. **Select File | New**, or *Ctrl + N*. With a blank file in front of us, we will use Emmet to generate our HTML. Type in the following Emmet string:

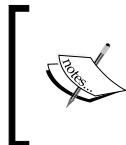
```
html>head+body>h3{index.html}+div#content
```

Now hit *Ctrl + Alt + Enter*, or from the **File menu**, select **Emmet | Expand Abbreviation**.

Voila! Emmet has generated the following HTML code in a millisecond - not bad for one line of source code:

```
<html>
<head></head>
<body>
  <h3>index.html</h3>
  <div id="content"></div>
</body>
</html>
```

Hit *Ctrl + S* to save the file, and enter `index.html`.



Only once we have saved a file, does Brackets start to do syntax highlighting based on the file extension. This is true of any Brackets file, so once you have created a file – TypeScript, CSS or HTML, save it to disk as soon as you can.

Back to Emmet.

Emmet uses the `>` character to create a child, and the `+` character to denote a sibling. If you specify curly braces `{ }` next to an element, this will be used as the text content.

The Emmet string that we entered previously basically said: "create an `html` tag with a child `head` tag. Then create another child tag of `html` named `body`, create a child `h3` tag with the text "`index.html`", and then create a sibling `div` tag as a child of `body` with the `id` of `content`." Definitely head over to <http://emmet.io> for further documentation, and remember to keep the cheat-sheet handy (<http://docs.emmet.io/cheat-sheet>), when you are learning Emmet string shortcuts.

Now lets finish off our `index.html` with an `app.js` script to load our TypeScript generated JavaScript file. Move your cursor in-between the `<head></head>` tags, and type another Emmet string:

```
script:src
```

Now hit `Ctrl + Alt + Enter`, to have Emmet generate a `<script src=""></script>` tag, and conveniently place your cursor between the quotes ready for you to simply fill in the blanks. Now type the JavaScript filename, `app.js`.

Your completed `index.html` file should now look as follows:

```
<html>
<head>
  <script src="app.js"></script>
</head>
<body>
  <h3>index.html</h3>
  <div id="content"></div>
</body>
</html>
```

This is all we need for our sample HTML page.

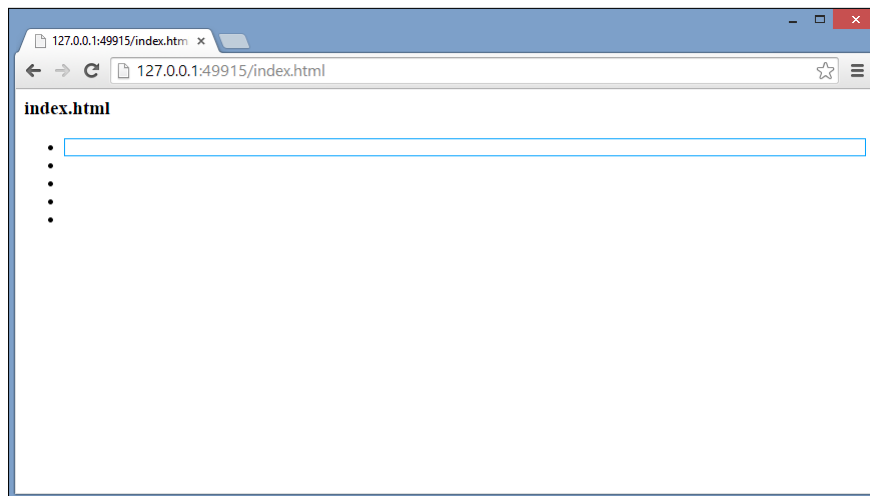
Using Brackets live preview

Within Brackets, click on the **live preview** icon on the far right of the screen - it's the electric zig zag one - just above the lego-block packages icon. This will launch Chrome and render our `index.html` in live preview mode. Just to show how Brackets can be used for live preview, keep this Chrome window visible, and navigate back to Brackets. You should be able to see both windows at the same time.

Now edit the `index.html` file, and type the following Emmet shorthand under your `<div id="content"></div>` element:

```
ul>li.item$*5
```

Again, hit `Ctrl + Alt + Enter`, and note how the generated `` and `` tags (5 of them) are automatically displayed in your Chrome browser. As you move your caret up or down in the source code, notice how the blue outline in Chrome shows the element in the web page.



Brackets running Chrome in live preview mode, showing highlighted elements

We won't be needing these `` `` tags for our application, so simply `Ctrl + Z`, `Ctrl + Z` to undo our changes, or delete the tags.

Creating a TypeScript file

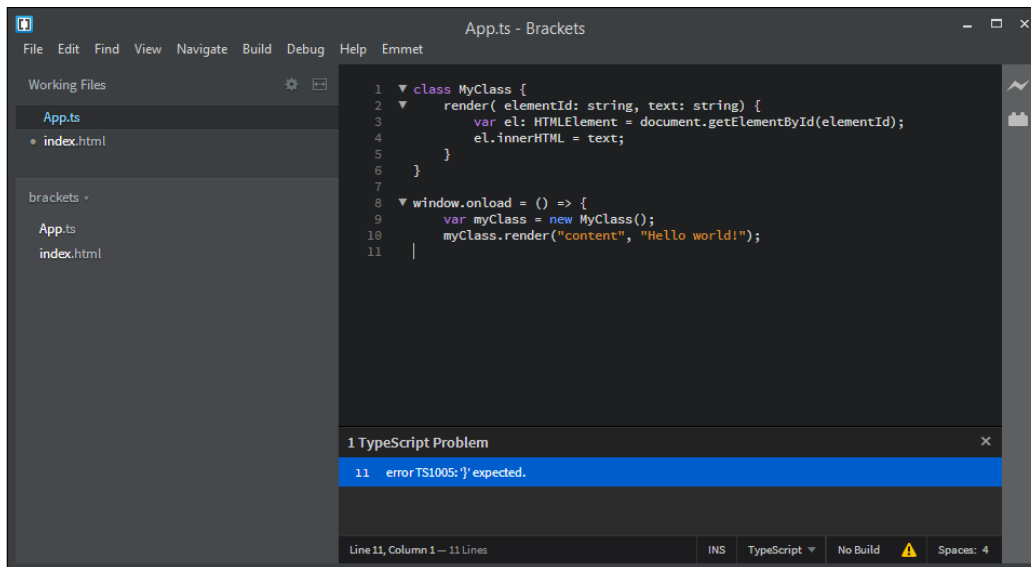
To create our very simple TypeScript application, hit `Ctrl + N` (new file), `Ctrl + S` (save file) and use `app.ts` as your file name. Start typing the following code, and notice how Brackets also does autocompletion, or Intellisense on the fly, similar to Visual Studio and WebStorm:

```
class MyClass {
  render( elementId: string, text: string) {
    var el: HTMLElement = document.getElementById(
      elementId);
    el.innerHTML = text;
  }
}
```

```
window.onload = () => {  
    var myClass = new MyClass();  
    myClass.render("content", "Hello world!");  
}
```

This is the same code that we used previously, and simply creates a TypeScript class named `MyClass` that has a single `render` function. This `render` function gets a DOM element, and modifies its `innerHTML` property. The `window.onload` function creates an instance of this class, then calls the `render` function with the appropriate parameters.

If you save the file by hitting `Ctrl + S` at any stage, Brackets will invoke the TypeScript language engine to verify our TypeScript, and render any errors in the bottom window pane. In the following screenshot, we can clearly see that we are missing a closing brace `}`.



Brackets editing a TypeScript file and showing compile errors

Brackets will not invoke the TypeScript compiler to generate an `app.js` file - it just parses the TypeScript code at this stage, and highlights any errors. Double-clicking on the error in the **TypeScript Problem** pane will jump to the line in question.

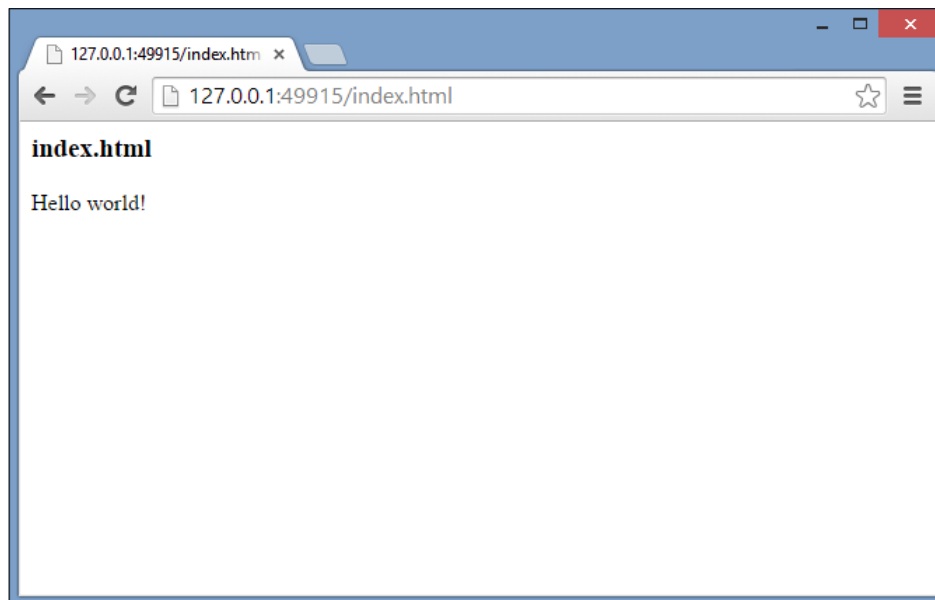
Compiling our TypeScript

Before we are able to run our application, we will need to compile the `app.ts` file into an `app.js` file by invoking the TypeScript compiler. Open up a Command Prompt, change to your source directory, and simply type:

```
tsc app.ts
```

This command will invoke the `tsc` command line compiler, and create an `app.js` file from our `app.ts` file.

Now that we have an `app.js` file in this directory, we can invoke the **live preview** button again, and now see that our TypeScript application has indeed rendered the **Hello world!** text as the `innerHTML` of the content `div`:



Brackets live preview running our TypeScript application

Using Grunt

Obviously, it is going to be very tedious to have to switch to the Command Prompt and manually compile each TypeScript file every time we have made a change. Grunt is an automated task runner (<http://gruntjs.com>) that can automate many tedious compile, build, and testing tasks. In this section, we will use Grunt to watch TypeScript files, and invoke the `tsc` compiler when a file is saved. This is very similar to WebStorm's file watch functionality that we used earlier.

Grunt runs in a Node environment. Node is an open-source, cross platform runtime environment, whose programs are written in JavaScript. To run Grunt, we will therefore need to install Node. Installers for Windows, Linux and OS X can be found from the Node website (<http://nodejs.org/>). Once Node is installed, we can use **npm (Node package manager)** to install Grunt and the Grunt command line interface.

Grunt needs to be installed as an npm dependency of your project. It cannot be installed globally, the way most npm packages can. In order to do this, we will need to create a `package.json` file in the root project. Open up a Command Prompt, and navigate to the root directory of your Brackets project. Then simply type:

```
npm init
```

And follow the prompts. You can pretty much leave all of the options as their default, and always go back to edit the `package.json` file that is created from this step, should you need to tweak any changes. With the package initialization step complete, we can now install Grunt as follows:

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

The `--save-dev` option will install a local version of Grunt in the project directory. This is done so that multiple projects on your machine can use different versions of Grunt. We will also need the `grunt-typescript` package, as well as the `grunt-contrib-watch` package. These can be installed with the following npm commands:

```
Npm install grunt-typescript --save-dev
```

```
Npm install grunt-contrib-watch --save-dev.
```

Lastly, we will need a `GruntFile.js` as the entry point for Grunt. Using Brackets, create a new file, save it as `GruntFile.js`, and enter the following JavaScript. Note that we are creating a JavaScript file here, not a TypeScript file. You can find a copy of this file in the sample source code that accompanies this chapter.

```
module.exports = function (grunt) {
  grunt.loadNpmTasks('grunt-typescript');
  grunt.loadNpmTasks('grunt-contrib-watch');
  grunt.initConfig({
    pkg: grunt.file.readJSON('package.json'),
    typescript: {
      base: {
        src: ['**/*.ts'],
        options: {
          module: 'commonjs',
          target: 'es5',
```

```
        sourceMap: true
      }
    },
  },
  watch: {
    files: '**/*.ts',
    tasks: ['typescript']
  }
});

//grunt.registerTask('default', ['typescript']);
grunt.registerTask('default', ['watch']);
}
```

This `Gruntfile.js` is necessary to setup all of the Grunt tasks. It is a simple function that Grunt uses to initialize the Grunt environment, and specify the Grunt commands. The first two lines of the function are loading `grunt-typescript` and `grunt-contrib-watch` tasks, and then runs the `grunt.initConfig` function with a configuration section. This configuration section has a `pkg` property, a `typescript` property and a `watch` property. The `pkg` property is set by reading the `package.json` file that we created earlier as part of the `npm init` step.

The `typescript` property has a `base` property, in which we are specifying that the source should be `'**/*.ts'` – in other words, all `.ts` files in any subdirectory. We are also specifying some TypeScript options – using `'commonjs'` modules instead of `'amd'` modules, and generating sourcemaps.

The `watch` property has two sub-properties. The `files` property specifies to watch for any `.ts` files in our source tree, and the `tasks` array specifies that we should kick off the TypeScript command once a file has been changed. Finally we call `grunt.registerTask`, specifying that the default task is to watch for file changes. Grunt will run in the background watching for saved files, and if found, will execute the TypeScript task.

We can now run Grunt from the command line. Make sure that you are in the Brackets project base directory, and fire up Grunt:

Grunt

Open up your `app.ts` file, make a small change (add a space or something), and then hit `Ctrl + S` to save. Now check back on the output from the Grunt command line. You should see something like this:

```
>> File "app.ts" changed.
Running "typescript:base" (typescript) task
2 files created. js: 1 file, map: 1 file, declaration: 0 files (861ms)
Done, without errors.
Completed in 1.665s at Fri Oct 10 2014 11:24:47 GMT+0800 (W. Australia
Standard Time) - Waiting...
```

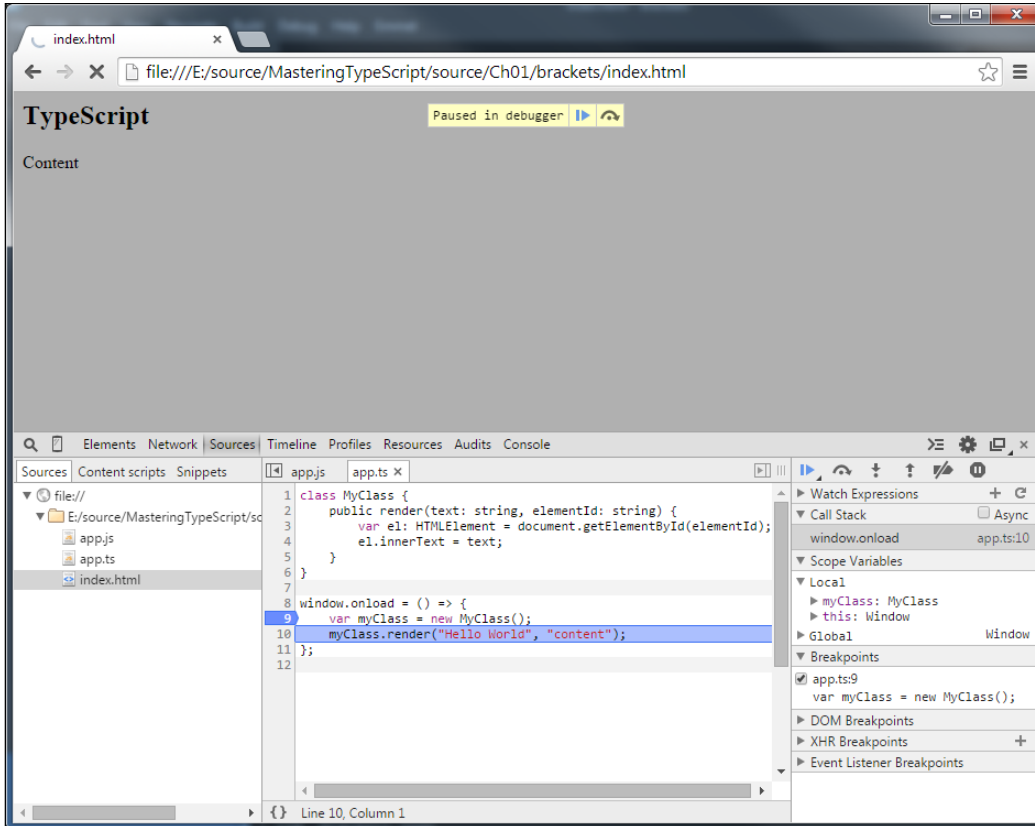
This command line output is confirmation that the Grunt watch task has identified `app.ts` has having changed, run the TypeScript task, created two files, and is now waiting for the next file to change. Flicking back to Brackets, we should now see the `app.js` file created by Grunt in the Brackets file pane.

Debugging in Chrome

Since Brackets is just being used as an editor, we will need to debug our applications using the standard Chrome development tools. One option that we specified in our `GruntFile.js` for TypeScript was to turn on sourcemaps (`options { sourceMap : true }`). With this option, Chrome – and other browsers – can map the running JavaScript back to the source TypeScript file. This means that you can set the debugger breakpoints in your TypeScript file, and walk through your TypeScript file while debugging.

To debug our sample app, firstly get the `index.html` page running in **Live Preview** mode, and hit `F12` to bring up the development tools. Chrome has a number of tools available for developers, including Network, Console, and Elements to inspect the DOM. Click on the **Sources** tab and hit `Ctrl + P` to open a file. Scroll down to `app.ts`, and hit `Enter`. Put a breakpoint on line 9 (`var myClass = new MyClass()`), and then re-load the page.

Chrome should pause the page in debugger mode as follows:



Brackets debugging TypeScript using Chrome development tools.

You can now use all of the Chrome debugging tools to your heart's content.

Summary

In this chapter we have had a quick look at what TypeScript is, and what benefits it can bring to the JavaScript development experience. We also looked at setting up a development environment using two popular commercial IDEs, and one open-source development environment. Now that we have a development environment setup, we can start looking at the TypeScript language itself in a bit more detail. We will start with types, move on to variables, and then discuss functions in the next chapter.

2

Types, Variables and Function Techniques

TypeScript introduces strong typing to JavaScript through a simple syntax, referred to by Anders Hejlsberg as "syntactic sugar".

This chapter is an introduction to the syntax used in the TypeScript language to apply strong typing to JavaScript. It is intended for readers that have not used TypeScript before, and covers the transition from standard JavaScript to TypeScript. If you already have experience with TypeScript, and have a good understanding of the topics listed below, then by all means have a quick read through, or skip to the next chapter.

We will cover the following topics in this chapter:

- Basic types and type syntax: strings, numbers, and booleans
- Inferred typing and duck-typing
- Arrays and enums
- The any type and explicit casting
- Functions and anonymous functions
- Optional and default function parameters
- Argument arrays
- Function callbacks and function signatures
- Function scoping rules and overloads

Basic types

JavaScript variables can hold a number of data types, including numbers, strings, arrays, objects, functions, and more. The type of an object in JavaScript is determined by its assignment – so if a variable has been assigned a string value, then it will be of type string. This can, however, introduce a number of problems in our code.

JavaScript is not strongly typed

As we saw in *Chapter 1, TypeScript – Tools and Framework Options*, JavaScript objects and variables can be changed or reassigned on the fly. As an example of this, consider the following JavaScript code:

```
var myString = "test";
var myNumber = 1;
var myBoolean = true;
```

We start by defining three variables, named `myString`, `myNumber` and `myBoolean`. The `myString` variable is set to a string value of "test", and as such will be of type string. Similarly, `myNumber` is set to the value of 1, and is therefore of type number, and `myBoolean` is set to `true`, making it of type boolean. Now let's start assigning these variables to each other, as follows:

```
myString = myNumber;
myBoolean = myString;
myNumber = myBoolean;
```

We start by setting the value of `myString` to the value of `myNumber` (which is the numeric value of 1). We then set the value of `myBoolean` to the value of `myString`, (which would now be the numeric value of 1). Finally, we set the value of `myNumber` to the value of `myBoolean`. What is happening here, is that even though we started out with three different types of variables – a string, a number, and a boolean – we are able to reassign any of these variables to one of the other types. We can assign a number to a string, a string to boolean, or a boolean to a number.

While this type of assignment in JavaScript is legal, it shows that the JavaScript language is not strongly typed. This can lead to unwanted behaviour in our code. Parts of our code may be relying on the fact that a particular variable is holding a string, and if we inadvertently assign a number to this variable, our code may start to break in unexpected ways.

TypeScript is strongly typed

TypeScript, on the other hand, is a strongly typed language. Once you have declared a variable to be of type `string`, you can only assign `string` values to it. All further code that uses this variable must treat it as though it has a type of `string`. This helps to ensure that code that we write will behave as expected. While strong typing may not seem to be of any use with simple strings and numbers – it certainly does become important when we apply the same rules to objects, groups of objects, function definitions and classes. If you have written a function that expects a `string` as the first parameter and a `number` as the second, you cannot be blamed, if someone calls your function with a `boolean` as the first parameter and something else as the second.

JavaScript programmers have always relied heavily on documentation to understand how to call functions, and the order and type of the correct function parameters. But what if we could take all of this documentation and include it within the IDE? Then, as we write our code, our compiler could point out to us – automatically – that we were using objects and functions in the wrong way. Surely this would make us more efficient, more productive programmers, allowing us to generating code with fewer errors?

TypeScript does exactly that. It introduces a very simple syntax to define the type of a variable or a function parameter to ensure that we are using these objects, variables, and functions in the correct manner. If we break any of these rules, the TypeScript compiler will automatically generate errors, pointing us to the lines of code that are in error.

This is how TypeScript got its name. It is JavaScript with strong typing - hence TypeScript. Let's take a look at this very simple language syntax that enables the "Type" in TypeScript.

Type syntax

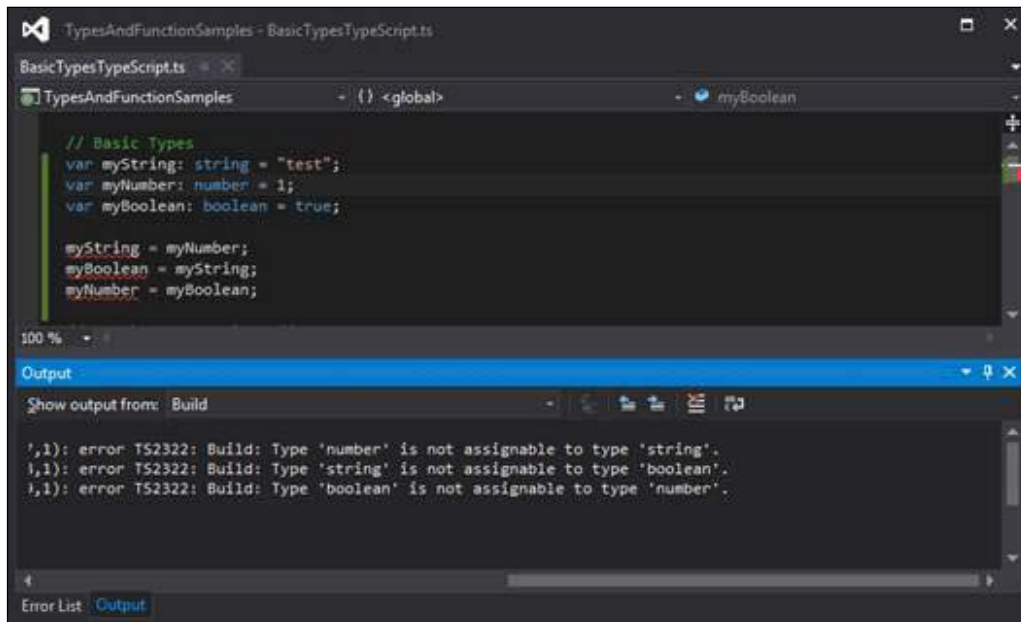
The TypeScript syntax for declaring the type of a variable is to include a colon (:), after the variable name, and then indicate its type. Consider the following TypeScript code:

```
var myString : string = "test";
var myNumber: number = 1;
var myBoolean : boolean = true;
```

This code snippet is the TypeScript equivalent of our preceding JavaScript code, and shows an example of the TypeScript syntax for declaring a type for the `myString` variable. By including a colon and then the keyword `string` (`: string`), we are telling the compiler that the `myString` variable is of type `string`. Similarly, the `myNumber` variable is of type `number`, and the `myBoolean` variable is of type `boolean`. TypeScript has introduced the `string`, `number` and `boolean` keywords for each of these basic JavaScript types.

If we attempt to assign a value to a variable that is not of the same type, the TypeScript compiler will generate a compile-time error. Given the variables declared in the preceding code, the following TypeScript code will generate some compile errors:

```
myString = myNumber;
myBoolean = myString;
myNumber = myBoolean;
```



TypeScript build errors when assigning incorrect types

The TypeScript compiler is generating compile errors, because we are attempting to mix these basic types. The first error is generated by the compiler because we cannot assign a `number` value to a variable of type `string`. Similarly, the second compile error indicates that we cannot assign a `string` value to a variable of type `boolean`. Again, the third error is generated because we cannot assign a `boolean` value to a variable of type `number`.

The strong typing syntax that the TypeScript language introduces, means that we need to ensure that the types on the left-hand side of an assignment operator (=) are the same as the types on the right-hand side of the assignment operator.

To fix the preceding TypeScript code, and remove the compile errors, we would need to do something similar to the following:

```
myString = myNumber.toString();
myBoolean = (myString === "test");
if (myBoolean) {
    myNumber = 1;
}
```

Our first line of code has been changed to call the `.toString()` function on the `myNumber` variable (which is of type `number`), in order to return a value that is of type `string`. This line of code, then, does not generate a compile error because both sides of the equal sign are of the same type.

Our second line of code has also been changed so that the right hand side of the assignment operator returns the result of a comparison, `myString === "test"`, which will return a value of type `boolean`. The compiler will therefore allow this code, because both sides of the assignment resolve to a value of type `boolean`.

The last line of our code snippet has been changed to only assign the value `1` (which is of type `number`) to the `myNumber` variable, if the value of the `myBoolean` variable is `true`.

Anders Hejlsberg describes this feature as "syntactic sugar". With a little sugar on top of comparable JavaScript code, TypeScript has enabled our code to conform to strong typing rules. Whenever you break these strong typing rules, the compiler will generate errors for your offending code.

Inferred typing

TypeScript also uses a technique called inferred typing, in cases where you do not explicitly specify the type of your variable. In other words, TypeScript will find the first usage of a variable within your code, figure out what type the variable is first initialized to, and then assume the same type for this variable in the rest of your code block. As an example of this, consider the following code:

```
var myString = "this is a string";
var myNumber = 1;
myNumber = myString;
```

We start by declaring a variable named `myString`, and assign a string value to it. TypeScript identifies that this variable has been assigned a value of type `string`, and will, therefore, infer any further usages of this variable to be of type `string`. Our second variable, named `myNumber` has a number assigned to it. Again, TypeScript is inferring the type of this variable to be of type `number`. If we then attempt to assign the `myString` variable (of type `string`) to the `myNumber` variable (of type `number`) in the last line of code, TypeScript will generate a familiar error message:

```
error TS2011: Build: Cannot convert 'string' to 'number'
```

This error is generated because of TypeScript's inferred typing rules.

Duck-typing

TypeScript also uses a method called duck-typing for more complex variable types. Duck-typing means that if it looks like a duck, and quacks like a duck, then it probably is a duck. Consider the following TypeScript code:

```
var complexType = { name: "myName", id: 1 };
complexType = { id: 2, name: "anotherName" };
```

We start with a variable named `complexType` that has been assigned a simple JavaScript object with a `name` and `id` property. On our second line of code, we can see that we are re-assigning the value of this `complexType` variable to another object that also has an `id` and a `name` property. The compiler will use duck-typing in this instance to figure out whether this assignment is valid. In other words, if an object has the same set of properties as another object, then they are considered to be of the same type.

To further illustrate this point, let's see how the compiler reacts if we attempt to assign an object to our `complexType` variable that does not conform to this duck-typing:

```
var complexType = { name: "myName", id: 1 };
complexType = { id: 2 };
complexType = { name: "anotherName" };
complexType = { address: "address" };
```

The first line of this code snippet defines our `complexType` variable, and assigns to it an object that contains both an `id` and `name` property. From this point, TypeScript will use this inferred type on any value we attempt to assign to the `complexType` variable. On our second line of code, we are attempting to assign a value that has an `id` property but not the `name` property. On the third line of code, we again attempt to assign a value that has a `name` property, but does not have an `id` property. On the last line of our code snippet, we have completely missed the mark. Compiling this code will generate the following errors:

```
error TS2012: Build: Cannot convert '{ id: number; }' to '{ name: string; id: number; }':
error TS2012: Build: Cannot convert '{ name: string; }' to '{ name: string; id: number; }':
error TS2012: Build: Cannot convert '{ address: string; }' to '{ name: string; id: number; }':
```

As we can see from the error messages, TypeScript is using duck-typing to ensure type safety. In each message, the compiler gives us clues as to what is wrong with the offending code - by explicitly stating what it is expecting. The `complexType` variable has both an `id` and a `name` property. To assign a value to the `complexType` variable, then, this value will need to have both an `id` and a `name` property. Working through each of these errors, TypeScript is explicitly stating what is wrong with each line of code.

Note that the following code will not generate any error messages:

```
var complexType = { name: "myName", id: 1 };
complexType = { name: "name", id: 2, address: "address" };
```

Again, our first line of code defines the `complexType` variable, as we have seen previously, with an `id` and a `name` property. Now, look at the second line of this example. The object we are using actually has three properties: `name`, `id`, and `address`. Even though we have added a new `address` property, the compiler will only check to see if our new object has both an `id` and a `name`. Because our new object has these properties, and will therefore match the original type of the variable, TypeScript will allow this assignment through duck-typing.

Inferred typing and duck-typing are powerful features of the TypeScript language - bringing strong typing to our code, without the need to use explicit typing, that is, a colon `:` and then the type specifier syntax.

Arrays

Besides the base JavaScript types of `string`, `number`, and `boolean`, TypeScript has two other data types: `Arrays` and `enums`. Let's look at the syntax for defining arrays.

An array is simply marked with the `[]` notation, similar to JavaScript, and each array can be strongly typed to hold a specific type as seen in the code below:

```
var arrayOfNumbers: number[] = [1, 2, 3];
arrayOfNumbers = [3, 4, 5];
arrayOfNumbers = ["one", "two", "three"];
```


On the first line of this code snippet, we are defining an array named `arrayOfNumbers`, and further specify that each element of this array must be of type `number`. The second line then reassigns this array to hold some different numerical values.

The last line of this snippet, however, will generate the following error message:

```
error TS2012: Build: Cannot convert 'string[]' to 'number[]':
```

This error message is warning us that the variable `arrayOfNumbers` is strongly typed to only accept values of type `number`. Our code tries to assign an array of strings to this array of numbers, and is therefore, generating a compile error.

The any type

All this type checking is well and good, but JavaScript is flexible enough to allow variables to be mixed and matched. The following code snippet is actually valid JavaScript code:

```
var item1 = { id: 1, name: "item 1" };
item1 = { id: 2 };
```

Our first line of code assigns an object with an `id` property and a `name` property to the variable `item1`. The second line then re-assigns this variable to an object that has an `id` property but not a `name` property. Unfortunately, as we have seen previously, TypeScript will generate a compile time error for the preceding code:

```
error TS2012: Build: Cannot convert '{ id: number; }' to '{ id: number; name: string; }'
```

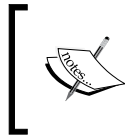
TypeScript introduces the `any` type for such occasions. Specifying that an object has a type of `any` in essence relaxes the compiler's strict type checking. The following code shows how to use the `any` type:

```
var item1 : any = { id: 1, name: "item 1" };
item1 = { id: 2 };
```

Note how our first line of code has changed. We specify the type of the variable `item1` to be of type `: any` so that our code will compile without errors. Without the type specifier of `: any`, the second line of code, would normally generate an error.

Explicit casting

As with any strongly typed language, there comes a time where you need to explicitly specify the type of an object. This concept will be expanded upon more thoroughly in the next chapter, but it is worthwhile to make a quick note of explicit casting here. An object can be cast to the type of another by using the `< >` syntax.



This is not a cast in the strictest sense of the word; it is more of an assertion that is used at runtime by the TypeScript compiler. Any explicit casting that you use will be compiled away in the resultant JavaScript and will not affect the code at runtime.

Let's modify our previous code snippet to use explicit casting:

```
var item1 = <any>{ id: 1, name: "item 1" };
item1 = { id: 2 };
```

Note that on the first line of this snippet, we have now replaced the `: any` type specifier on the left hand side of the assignment, with an explicit cast of `<any>` on the right hand side. This snippet of code is telling the compiler to explicitly cast, or to explicitly treat the `{ id: 1, name: "item 1" }` object on the right-hand side as a type of `any`. So the `item1` variable, therefore, also has the type of `any` (due to TypeScript's inferred typing rules). This then allows us to assign an object with only the `{ id: 2 }` property to the variable `item1` on the second line of code. This technique of using the `< >` syntax on the right hand side of an assignment, is called explicit casting.

While the `any` type is a necessary feature of the TypeScript language – its usage should really be limited as much as possible. It is a language shortcut that is necessary to ensure compatibility with JavaScript, but over-use of the `any` type will quickly lead to coding errors that will be difficult to find. Rather than using the type `any`, try to figure out the correct type of the object you are using, and then use this type instead. We use an acronym within our programming teams: **S.F.I.A.T.** (pronounced *sviat* or *sveat*). **Simply Find an Interface for the Any Type.** While this may sound silly – it brings home the point that the `any` type should always be replaced with an interface – so simply find it. An interface is a way of defining custom types in TypeScript, and we will cover interfaces in the next chapter. Just remember that by actively trying to define what an object's type should be, we are building strongly typed code, and therefore protecting ourselves from future coding errors and bugs.

Enums

Enums are a special type that has been borrowed from other languages such as C#, and provide a solution to the problem of special numbers. An enum associates a human-readable name for a specific number. Consider the following code:

```
enum DoorState {
    Open,
    Closed,
    Ajar
}
```

In this code snippet, we have defined an enum called `DoorState` to represent the state of a door. Valid values for this door state are `Open`, `Closed`, or `Ajar`. Under the hood (in the generated JavaScript), TypeScript will assign a numeric value to each of these human-readable enum values. In this example, the `DoorState.Open` enum value will equate to a numeric value of 0. Likewise, the enum value `DoorState.Closed` will be equate to the numeric value of 1, and the `DoorState.Ajar` enum value will equate to 2. Let's have a quick look at how we would use these enum values:

```
window.onload = () => {
  var myDoor = DoorState.Open;
  console.log("My door state is " + myDoor.toString());
};
```

The first line within the `window.onload` function creates a variable named `myDoor`, and sets its value to `DoorState.Open`. The second line simply logs the value of `myDoor` to the console. The output of this `console.log` function would be:

My door state is 0

This clearly shows that the TypeScript compiler has substituted the enum value of `DoorState.Open` with the numeric value 0. Now let's use this enum in a slightly different way:

```
window.onload = () => {
  var openDoor = DoorState["Closed"];
  console.log("My door state is " + openDoor.toString());
};
```

This code snippet uses a string value of "Closed" to lookup the enum type, and assign the resulting enum value to the `openDoor` variable. The output of this code would be:

My door state is 1

This sample clearly shows that the enum value of `DoorState.Closed` is the same as the enum value of `DoorState["Closed"]`, because both variants resolve to the numeric value of 1. Finally, let's have a look at what happens when we reference an enum using an array type syntax:

```
window.onload = () => {
  var ajarDoor = DoorState[2];
  console.log("My door state is " + ajarDoor.toString());
};
```

Here, we assign the variable `openDoor` to an enum value based on the 2nd index value of the `DoorState` enum. The output of this code, though, is surprising:

My door state is Ajar

You may have been expecting the output to be simply `2`, but here we are getting the string `"Ajar"` – which is a string representation of our original enum name. This is actually a neat little trick – allowing us to access a string representation of our enum value. The reason that this is possible is down to the JavaScript that has been generated by the TypeScript compiler. Let's have a look, then, at the closure that the TypeScript compiler has generated:

```
var DoorState;
(function (DoorState) {
    DoorState[DoorState["Open"] = 0] = "Open";
    DoorState[DoorState["Closed"] = 1] = "Closed";
    DoorState[DoorState["Ajar"] = 2] = "Ajar";
})(DoorState || (DoorState = {}));
```

This strange looking syntax is building an object that has a specific internal structure. It is this internal structure that allows us to use this enum in the various ways that we have just explored. If we interrogate this structure while debugging our JavaScript, we will see the internal structure of the `DoorState` object is as follows:

```
DoorState
{...}
  [prototype]: {...}
  [0]: "Open"
  [1]: "Closed"
  [2]: "Ajar"
  [prototype]: []
  Ajar: 2
  Closed: 1
  Open: 0
```

The `DoorState` object has a property called `"0"`, which has a string value of `"Open"`. Unfortunately, in JavaScript the number `0` is not a valid property name, so we cannot access this property by simply using `DoorState.0`. Instead, we must access this property using either `DoorState[0]` or `DoorState["0"]`. The `DoorState` object also has a property named `Open`, which is set to the numeric value `0`. The word `Open` IS a valid property name in JavaScript, so we can access this property using `DoorState["Open"]`, or simply `DoorState.Open`, which equate to the same property in JavaScript.

While the underlying JavaScript can be a little confusing, all we need to remember about enums is that they are a handy way of defining an easily remembered, human-readable name to a special number. Using human-readable enums, instead of just scattering various special numbers around in our code, also makes the intent of the code clearer. Using an application wide value named `DoorState.Open` or `DoorState.Closed` is far simpler than remembering to set a value to 0 for `Open`, 1 for `Closed`, and 3 for `ajar`. As well as making our code more readable, and more maintainable, using enums also protects our code base whenever these special numeric values change - because they are all defined in one place.

One last note on enums - we can set the numeric value manually, if needs be:

```
enum DoorState {
    Open = 3,
    Closed = 7,
    Ajar = 10
}
```

Here, we have overridden the default values of the enum to set `DoorState.Open` to 3, `DoorState.Closed` to 7, and `DoorState.Ajar` to 10.

Const enums

With the release of TypeScript 1.4, we are also able to define `const` enums as follows:

```
const enum DoorStateConst {
    Open,
    Closed,
    Ajar
}

var myState = DoorStateConst.Open;
```

These types of enums have been introduced largely for performance reasons, and the resultant JavaScript will not contain the full closure definition for the `DoorStateConst` enum as we saw previously. Let's have a quick look at the JavaScript that is generated from this `DoorStateConst` enum:

```
var myState = 0 /* Open */;
```

Note how we do not have a full JavaScript closure for the `DoorStateConst` enum at all. The compiler has simply resolved the `DoorStateConst.Open` enum to its internal value of 0, and removed the `const enum` definition entirely.

With `const` enums, we therefore cannot reference the internal string value of an enum, as we did in our previous code sample. Consider the following example:

```
// generates an error
console.log(DoorStateConst[0]);
// valid usage
console.log(DoorStateConst["Open"]);
```

The first `console.log` statement will now generate a compile time error – as we do not have the full closure available with the property of `[0]` for our `const` enum. The second usage of this `const` enum is valid, however, and will generate the following JavaScript:

```
console.log(0 /* "Open" */);
```

When using `const` enums, just keep in mind that the compiler will strip away all enum definitions and simply substitute the numeric value of the enum directly into our JavaScript code.

Functions

JavaScript defines functions using the `function` keyword, a set of braces, and then a set of curly braces. A typical JavaScript function would be written as follows:

```
function addNumbers(a, b) {
    return a + b;
}

var result = addNumbers(1, 2);
var result2 = addNumbers("1", "2");
```

This code snippet is fairly self-explanatory; we have defined a function named `addNumbers` that takes two variables and returns their sum. We then invoke this function, passing in the values of 1 and 2. The value of the variable `result` would then be `1 + 2`, which is 3. Now have a look at the last line of code. Here, we are invoking the `addNumbers` function, passing in two strings as arguments, instead of numbers. The value of the variable `result2` would then be a string, `"12"`. This string value seems like it may not be the desired result, as the name of the function is `addNumbers`.

Copying the preceding code into a TypeScript file would not generate any errors, but let's insert some type rules to the preceding JavaScript to make it more robust:

```
function addNumbers(a: number, b: number): number {
    return a + b;
};

var result = addNumbers(1, 2);
var result2 = addNumbers("1", "2");
```

In this TypeScript code, we have added a `:number` type to both of the parameters of the `addNumbers` function (`a` and `b`), and we have also added a `:number` type just after the `()` braces. Placing a type descriptor here means that the return type of the function itself is strongly typed to return a value of type `number`. In TypeScript, the last line of code, however, will cause a compilation error:

error TS2082: Build: Supplied parameters do not match any signature of call target:

This error message is generated because we have explicitly stated that the function should accept only numbers for both of the arguments `a` and `b`, but in our offending code, we are passing two strings. The TypeScript compiler, therefore, cannot match the signature of a function named `addNumbers` that accepts two arguments of type `string`.

Anonymous functions

The JavaScript language also has the concept of anonymous functions. These are functions that are defined on the fly and don't specify a function name. Consider the following JavaScript code:

```
var addVar = function(a, b) {
    return a + b;
};

var result = addVar(1, 2);
```

This code snippet defines a function that has no name and adds two values. Because the function does not have a name, it is known as an anonymous function. This anonymous function is then assigned to a variable named `addVar`. The `addVar` variable, then, can then be invoked as a function with two parameters, and the return value will be the result of executing the anonymous function. In this case, the variable `result` will have a value of 3.

Let's now rewrite the preceding JavaScript function in TypeScript, and add some type syntax, in order to ensure that the function only accepts two arguments of type `number`, and returns a value of type `number`:

```
var addVar = function(a: number, b: number): number {
    return a + b;
}

var result = addVar(1, 2);
var result2 = addVar("1", "2");
```

In this code snippet, we have created an anonymous function that accepts only arguments of type `number` for the parameters `a` and `b`, and also returns a value of type `number`. The types for both the `a` and `b` parameters, as well as the return type of the function, are now using the `:number` syntax. This is another example of the simple "syntactic sugar" that TypeScript injects into the language. If we compile this code, TypeScript will reject the code on the last line, where we try to call our anonymous function with two string parameters:

```
error TS2082: Build: Supplied parameters do not match any signature of call target:
```

Optional parameters

When we call a JavaScript function that has is expecting parameters, and we do not supply these parameters, then the value of the parameter within the function will be undefined. As an example of this, consider the following JavaScript code:

```
var concatStrings = function(a, b, c) {
    return a + b + c;
}

console.log(concatStrings("a", "b", "c"));
console.log(concatStrings("a", "b"));
```

Here, we have defined a function called `concatStrings` that takes three parameters, `a`, `b`, and `c`, and simply returns the sum of these values. If we call this function with all three parameters, as seen in the second last line of this snippet, we will end up with the string `"abc"` logged to the console. If, however, we only supply two parameters, as seen in the last line of this snippet, the string `"abundefined"` will be logged to the console. Again, if we call a function and do not supply a parameter, then this parameter, `c` in our case, will be simply `undefined`.

TypeScript introduces the question mark `?` syntax to indicate optional parameters. Consider the following TypeScript function definition:

```
var concatStrings = function(a: string, b: string, c?: string) {  
    return a + b + c;  
}  
  
console.log(concatStrings("a", "b", "c"));  
console.log(concatStrings("a", "b"));  
console.log(concatStrings("a"));
```

This is a strongly typed version of the original `concatStrings` JavaScript function that we were using previously. Note the addition of the `?` character in the syntax for the third parameter: `c?: string`. This indicates that the third parameter is optional, and therefore, all of the preceding code will compile cleanly, except for the last line. The last line will generate an error:

```
error TS2081: Build: Supplied parameters do not match any signature of call target.
```

This error is generated because we are attempting to call the `concatStrings` function with only a single parameter. Our function definition, though, requires at least two parameters, with only the third parameter being optional.



The optional parameters must be the last parameters in the function definition. You can have as many optional parameters as you want, as long as non-optional parameters precede the optional parameters.

Default parameters

A subtle variant on the optional parameter function definition, allows us to specify the default value of a parameter if it is not passed in as an argument from the calling code. Let's modify our preceding function definition to use an optional parameter:

```
var concatStrings = function(a: string, b: string, c: string = "c") {  
    return a + b + c;  
}  
  
console.log(concatStrings("a", "b", "c"));  
console.log(concatStrings("a", "b"));
```

This function definition has now dropped the `?` optional parameter syntax, but instead has assigned a value of `"c"` to the last parameter: `c:string = "c"`. By using default parameters, if we do not supply a value for the final parameter named `c`, the `concatStrings` function will substitute the default value of `"c"` instead. The argument `c`, therefore, will not be undefined. The output of the last two lines of code will both be `"abc"`.



Note that using the default parameter syntax will automatically make the parameter optional.

The arguments variable

The JavaScript language allows a function to be called with a variable number of arguments. Every JavaScript function has access to a special variable, named `arguments`, that can be used to retrieve all arguments that have been passed into the function. As an example of this, consider the following JavaScript code:

```
function testParams() {
  if (arguments.length > 0) {
    for (var i = 0; i < arguments.length; i++) {
      console.log("Argument " + i + " = " + arguments[i]);
    }
  }
}

testParams(1, 2, 3, 4);
testParams("first argument");
```

In this code snippet, we have defined a function name `testParams` that does not have any named parameters. Note, though, that we can use the special variable, named `arguments`, to test whether the function was called with any arguments. In our sample, we can simply loop through the `arguments` array, and log the value of each argument to the console, by using an array indexer: `arguments[i]`. The output of the `console.log` calls are as follows:

```
Argument 0 = 1
Argument 1 = 2
Argument 2 = 3
Argument 3 = 4
Argument 0 = first argument
```


So, how do we express a variable number of function parameters in TypeScript? The answer is to use what are called rest parameters, or the three dots (...) syntax. Here is the equivalent `testParams` function, expressed in TypeScript:

```
function testParams(...argArray: number[]) {
  if (argArray.length > 0) {
    for (var i = 0; i < argArray.length; i++) {
      console.log("argArray " + i + " = " + argArray[i]);
      console.log("arguments " + i + " = " + arguments[i]);
    }
  }
}

testParams(1);
testParams(1, 2, 3, 4);
testParams("one", "two");
```

Note the use of the `...argArray: number[]` syntax for our `testParams` function. This syntax is telling the TypeScript compiler that the function can accept any number of arguments. This means that our usages of this function, i.e. calling the function with either `testParams(1)` or `testParams(1, 2, 3, 4)`, will both compile correctly. In this version of the `testParams` function, we have added two `console.log` lines, just to show that the `arguments` array can be accessed by either the named rest parameter, `argArray[i]`, or through the normal JavaScript array, `arguments[i]`.

The last line in this sample will, however, generate a compile error, as we have defined the rest parameter to only accept numbers, and we are attempting to call the function with strings.

 The subtle difference between using `argArray` and `arguments` is the inferred type of the argument. Since we have explicitly specified that `argArray` is of type `number`, TypeScript will treat any item of the `argArray` array as a `number`. However, the internal `arguments` array does not have an inferred type, and so will be treated as the any type.

We can also combine normal parameters along with rest parameters in a function definition, as long as the rest parameters are the last to be defined in the parameter list, as follows:

```
function testParamsTs2(arg1: string,
  arg2: number, ...ArgArray: number[]) {
}
```

Here, we have two normal parameters named `arg1` and `arg2` and then an `argArray` rest parameter. Mistakenly placing the rest parameter at the beginning of the parameter list will generate a compile error.

Function callbacks

One of the most powerful features of JavaScript—and in fact the technology that Node was built on—is the concept of callback functions. A callback function is a function that is passed into another function. Remember that JavaScript is not strongly typed, so a variable can also be a function. This is best illustrated by having a look at some JavaScript code:

```
function myCallBack(text) {
    console.log("inside myCallback " + text);
}

function callingFunction(initialText, callback) {
    console.log("inside CallingFunction");
    callback(initialText);
}

callingFunction("myText", myCallBack);
```

Here, we have a function named `myCallBack` that takes a parameter and logs its value to the console. We then define a function named `callingFunction` that takes two parameters: `initialText` and `callback`. The first line of this function simply logs "inside CallingFunction" to the console. The second line of the `callingFunction` is the interesting bit. It assumes that the `callback` argument is in fact a function, and invokes it. It also passes the `initialText` variable to the callback function. If we run this code, we will get two messages logged to the console, as follows:

```
inside CallingFunction
inside myCallback myText
```

But what happens if we do not pass a function as a callback? There is nothing in the preceding code that signals to us that the second parameter of `callingFunction` must be a function. If we inadvertently called the `callingFunction` function with a string, instead of a function as the second parameter, as follows:

```
callingFunction("myText", "this is not a function");
```

We would get a JavaScript runtime error:

```
0x800a138a - JavaScript runtime error: Function expected
```

Defensive minded programmers, however, would first check whether the callback parameter was in fact a function before invoking it, as follows:

```
function callingFunction(initialText, callback) {
  console.log("inside CallingFunction");
  if (typeof callback === "function") {
    callback(initialText);
  } else {
    console.log(callback + " is not a function");
  }
}

callingFunction("myText", "this is not a function");
```

Note the third line of this code snippet, where we check the type of the callback variable before invoking it. If it is not a function, we then log a message to the console. On the last line of this snippet, we are executing the `callingFunction`, but this time passing a string as the second parameter.

The output of the code snippet would be:

```
inside CallingFunction
this is not a function is not a function
```

When using function callbacks, then, JavaScript programmers need to do two things; firstly, understand which parameters are in fact callbacks and secondly, code around the invalid use of callback functions.

Function signatures

The TypeScript "syntactic sugar" that enforces strong typing, is not only intended for variables and types, but for function signatures as well. What if we could document our JavaScript callback functions in code, and then warn users of our code when they are passing the wrong type of parameter to our functions ?

TypeScript does this through function signatures. A function signature introduces a fat arrow syntax, `() =>`, to define what the function should look like. Let's re-write the preceding JavaScript sample in TypeScript:

```
function myCallBack(text: string) {
  console.log("inside myCallback " + text);
}
```

```
function callingFunction(initialText: string,
    callback: (text: string) => void)
{
    callback(initialText);
}

callingFunction("myText", myCallBack);
callingFunction("myText", "this is not a function");
```

Our first function definition, `myCallBack` now strongly types the `text` parameter to be of type `string`. Our `callingFunction` function has two parameters; `initialText`, which is of type `string`, and `callback`, which now has the new function signature syntax. Let's look at this function signature more closely:

```
callback: (text: string) => void
```

What this function definition is saying, is that the `callback` argument is typed (by the `:` syntax) to be a function, using the fat arrow syntax `() =>`. Additionally, this function takes a parameter named `text` that is of type `string`. To the right of the fat arrow syntax, we can see a new TypeScript basic type, called `void`. `Void` is a keyword to denote that a function does not return a value.

So, the `callingFunction` function will only accept, as its second argument, a function that takes a single `string` parameter and returns nothing. Compiling the preceding code will correctly highlight an error in the last line of the code snippet, where we passing a `string` as the second parameter, instead of a callback function:

```
error TS2082: Build: Supplied parameters do not match any signature of call target:
Type '(text: string) => void' requires a call signature, but type 'String' lacks one
```

Given the preceding function signature for the callback function, the following code would also generate compile time errors:

```
function myCallBackNumber(arg1: number) {
    console.log("arg1 = " + arg1);
}

callingFunction("myText", myCallBackNumber);
```

Here, we are defining a function named `myCallBackNumber`, that takes a `number` as its only parameter. When we attempt to compile this code, we will get an error message indicating that the `callback` parameter, which is our `myCallBackNumber` function, also does not have the correct function signature:

```
Call signatures of types 'typeof myCallBackNumber' and '(text: string) => void' are incompatible.
```

The function signature of `myCallbackNumber` would actually be `(arg1: number) => void`, instead of the required `(text: string) => void`, hence the error.



In function signatures, the parameter name (`arg1` or `text`) does not need to be the same. Only the number of parameters, their types, and the return type of the function need to be the same.

This is a very powerful feature of TypeScript — defining in code what the signatures of functions should be, and warning users when they do not call a function with the correct parameters. As we saw in our introduction to TypeScript, this is most significant when we are working with third-party libraries. Before we are able to use third-party functions, classes, or objects in TypeScript, we need to define what their function signatures are. These function definitions are put into a special type of TypeScript file, called a declaration file, and saved with a `.d.ts` extension. We will take an in-depth look at declaration files in *Chapter 4, Writing and Using Declaration Files*.

Function callbacks and scope

JavaScript uses lexical scoping rules to define the valid scope of a variable. This means that the value of a variable is defined by its location within the source code. Nested functions have access to variables that are defined in their parent scope. As an example of this, consider the following TypeScript code:

```
function testScope() {
    var testVariable = "myTestVariable";
    function print() {
        console.log(testVariable);
    }
}

console.log(testVariable);
```

This code snippet defines a function named `testScope`. The variable `testVariable` is defined within this function. The `print` function is a child function of `testScope`, so it has access to the `testVariable` variable. The last line of the code, however, will generate a compile error, because it is attempting to use the variable `testVariable`, which is lexically scoped to be valid only inside the body of the `testScope` function:

```
error TS2095: Build: Could not find symbol 'testVariable'.
```

Simple, right? A nested function has access to variables depending on its location within the source code. This is all well and good, but in large JavaScript projects, there are many different files and many areas of the code are designed to be re-usable.

Let's take a look at how these scoping rules can become a problem. For this sample, we will use a typical callback scenario – using jQuery to execute an asynchronous call to fetch some data. Consider the following TypeScript code:

```
var testVariable = "testValue";

function getData() {
    var testVariable_2 = "testValue_2";
    $.ajax(
        {
            url: "/sample_json.json",
            success: (data, status, jqXHR) => {
                console.log("success : testVariable is "
                    + testVariable);
                console.log("success : testVariable_2 is"
                    + testVariable_2);
            },
            error: (message, status, stack) => {
                alert("error " + message);
            }
        }
    );
}

getData();
```

In this code snippet, we are defining a variable named `testVariable` and setting its value. We then define a function called `getData`. The `getData` function sets another variable called `testVariable_2`, and then calls the jQuery `$.ajax` function. The `$.ajax` function is configured with three properties: `url`, `success`, and `error`. The `url` property is a simple string that points to a `sample_json.json` file in our project directory. The `success` property is an anonymous function callback, that simply logs the values of `testVariable` and `testVariable_2` to the console. Finally, the `error` property is also an anonymous function callback, that simply pops up an alert.

This code runs as expected, and the success function will log the following results to the console:

```
success : testVariable is :testValue
success : testVariable_2 is :testValue_2
```


So far so good. Now, let's assume that we are trying to refactor the preceding code, as we are doing quite a few similar `$.ajax` calls, and want to reuse the `success` callback function elsewhere. We can easily switch out this anonymous function, and create a named function for our `success` callback, as follows:

```
var testVariable = "testValue";

function getData() {
  var testVariable_2 = "testValue_2";
  $.ajax(
    {
      url: "/sample_json.json",
      success: successCallback,
      error: (message, status, stack) => {
        alert("error " + message);
      }
    }
  );
}

function successCallback(data, status, jqXHR) {
  console.log("success : testVariable is :" + testVariable);
  console.log("success : testVariable_2 is :" + testVariable_2);
}

getData();
```

In this sample, we have created a new function named `successCallback` with the same parameters as our previous anonymous function. We have also modified the `$.ajax` call to simply pass this function in, as a callback function for the `success` property: `success: successCallback`. If we were to compile this code now, TypeScript would generate an error, as follows:

```
error TS2095: Build: Could not find symbol 'testVariable_2'.
```

Since we have changed the lexical scope of our code, by creating a named function, the new `successCallback` function no longer has access the variable `testVariable_2`.



It is fairly easy to spot this sort of error in a trivial example, but in larger projects, and when using third-party libraries, these sorts of errors become more difficult to track down. It is, therefore, worth mentioning that when using callback functions, we need to understand this lexical scope. If your code expects a property to have a value, and it does not have one after a callback, then remember to have a look at the context of the calling code.

Function overloads

As JavaScript is a dynamic language, we can often call the same function with different argument types. Consider the following JavaScript code:

```
function add(x, y) {
    return x + y;
}

console.log("add(1,1)=" + add(1,1));
console.log("add('1','1')=" + add("1", "1"));
console.log("add(true,false)=" + add(true, false));
```

Here, we are defining a simple `add` function that returns the sum of its two parameters, `x` and `y`. The last three lines of this code snippet simply log the result of the `add` function with different types: two numbers, two strings, and two boolean values. If we run this code, we will see the following output:

```
add(1,1)=2
add('1','1')=11
add(true,false)=1
```

TypeScript introduces a specific syntax to indicate multiple function signatures for the same function. If we were to replicate the preceding code in TypeScript, we would need to use the function overload syntax:

```
function add(arg1: string, arg2: string): string;
function add(arg1: number, arg2: number): number;
function add(arg1: boolean, arg2: boolean): boolean;
function add(arg1: any, arg2: any): any {
    return arg1 + arg2;
}

console.log("add(1,1)=" + add(1, 1));
console.log("add('1','1')=" + add("1", "1"));
console.log("add(true,false)=" + add(true, false));
```

The first line of this code snippet specifies a function overload signature for the `add` function that accepts two strings and returns a `string`. The second line specifies another function overload that uses numbers, and the third line uses booleans. The fourth line contains the actual body of the function and uses the type specifier of `any`. The last three lines of this snippet show how we would use these function signatures, and are similar to the JavaScript code that we have been using previously.

There are three points of interest in the preceding code snippet. Firstly, none of the function signatures on the first three lines of the snippet actually have a function body. Secondly, the final function definition uses the type specifier of `any` and eventually includes the function body. The function overload syntax must follow this structure, and the final function signature, that includes the body of the function must use the `any` type specifier, as anything else will generate compile-time errors.

The third point to note, is that we are limiting the `add` function, by using these function overload signatures, to only accept two parameters that are of the same type. If we were to try and mix our types; for example, if we call the function with a `boolean` and a `string`, as follows:

```
console.log("add(true, '1')", add(true, "1"));
```

TypeScript would generate compile errors:

```
error TS2082: Build: Supplied parameters do not match any signature of call target:
error TS2087: Build: Could not select overload for 'call' expression.
```

This seems to contradict our final function definition though. In the original TypeScript sample, we had a function signature that accepted `(arg1: any, arg2: any)`; so, in theory, this should be called when we try to add a `boolean` and a `number`. The TypeScript syntax for function overloads, however, does not allow this. Remember that the function overload syntax must include the use of the `any` type for the function body, as all overloads eventually call this function body. However, the inclusion of the function overloads above the function body indicates to the compiler that these are the only signatures that should be available to the calling code.

Union types

With the release of TypeScript 1.4, we now have the ability to combine one or two types using the pipe symbol (`|`) to denote a Union Type. We can, therefore, rewrite our `add` function overrides in the previous code snippet as follows:

```
function addWithUnion(
  arg1: string | number | boolean,
  arg2: string | number | boolean
): string | number | boolean
{
  if (typeof arg1 === "string") {
    // arg1 is treated as a string here
    return arg1 + "is a string";
  }
}
```

```
    if (typeof arg1 === "number") {
        // arg1 is treated as a number here
        return arg1 + 10;
    }
    if (typeof arg1 === "boolean") {
        // arg1 is treated as a boolean here
        return arg1 && false;
    }
}
```

This function, named `addWithUnion` has two arguments, `arg1` and `arg2`. These arguments are now using the union type syntax to specify that these arguments can be either `string`, `number`, or `boolean`. Notice too that our return type for the function is again using union types, meaning that the function will return one of these types as well.

Type guards

Within the body of the `addWithUnion` function in the preceding code snippet, we check whether the type of the `arg1` argument is a string, with the statement `typeof arg1 === "string"`. This is known as a type guard and means that the type of `arg1` will be treated as a `string` within the `if` statement block. Within the body of the next `if` statement, the type of `arg1` will be treated as a `number`, allowing us to add 10 to its value, and in the body of the last `if` statement, the type will be treated as a `boolean` by the compiler.

Type aliases

We are also able to define an alias for a type, a union type, or a function definition. Type aliases are denoted by using the `type` keyword. We can, therefore, write our preceding `add` function as follows:

```
type StringNumberOrBoolean = string | number | boolean;

function addWithAliases(
    arg1: StringNumberOrBoolean,
    arg2: StringNumberOrBoolean
): StringNumberOrBoolean {

}
```

Here, we have defined a type alias named `StringNumberOrBoolean` that is a type union of the `string`, `number`, and `boolean` types.

Type aliases can also be used for function signatures as follows:

```
type CallbackWithString = (string) => void;

function usingCallback(callback: CallbackWithString) {
    callback("this is a string");
}
```

Here, we have defined a type alias named `CallbackWithString` that is a function that takes a single `string` parameter and returns a `void`. Our `usingCallback` function accepts this type alias within the function signature as the type for the `callback` argument.

Summary

In this chapter, we have discussed TypeScript's basic types, variables, and function techniques. We saw how TypeScript introduces "syntactic sugar" on top of normal JavaScript code, to ensure strongly typed variables and function signatures. We also saw how TypeScript uses duck-typing and explicit casting, and finished up with a discussion on TypeScript functions, function signatures, and overloading. In the next chapter, we will build on this knowledge and see how TypeScript extends these strongly typed rules into interfaces, classes and generics.

3

Interfaces, Classes and Generics

We have already seen how TypeScript uses basic types, inferred types, and function signatures to bring a strongly typed development experience to JavaScript. TypeScript also introduces three concepts borrowed from other object-oriented languages: interfaces, classes and generics. In this chapter, we will look at these object-oriented concepts, how they are used in TypeScript, and what benefits they bring to JavaScript programmers.

The first section of this chapter is intended for readers that are using TypeScript for the first time, and covers interfaces, classes and inheritance from the ground up. The second section of this chapter builds on this knowledge, and shows how to create and use the Factory Design Pattern. The third section of this chapter deals with generics.

If you have experience with TypeScript, are actively using interfaces and classes, understand inheritance, and are comfortable with the lexical scoping rules as applied to the `this` parameter, then you may be more interested in the later sections on the Factory Design Pattern, or generics.

This chapter will cover the following topics:

- Interfaces
- Classes
- Inheritance
- Closures
- The Factory Design Pattern
- Class modifiers, static functions and properties
- Generics
- Runtime type checking

Interfaces

An interface provides us with a mechanism to define what properties and methods an object must implement. If an object adheres to an interface, it is said that the object implements the interface. TypeScript will generate compile errors earlier in our code if an object does not implement an interface properly. The interface is also another way of defining a custom type, and gives us, among other things, an early indication – at the time we are constructing an object – that the object does not have the properties and methods that we require.

Consider the following TypeScript code:

```
interface IComplexType {
    id: number;
    name: string;
}

var complexType : IComplexType =
    { id: 1, name: "firstObject" };
var complexType_2: IComplexType =
    { id: 2, description: "myDescription"};

if (complexType == complexType_2) {
    console.log("types are equal");
}
```

We start with an interface named `IComplexType` that has an `id` and a `name` property. The `id` property is strongly typed to be of type `number`, and the `name` property is of type `string`. We then create a variable named `complexType`, and use the `:` type syntax to indicate that this variable is of type `IComplexType`. The next variable, named `complexType_2`, also strongly types this variable to be of type `IComplexType`. We then compare the `complexType` and `complexType_2` variables, and log a message to the console if these objects are the same. This code, however, will generate a compile error:

```
error TS2012: Build: Cannot convert
' { id: number; description: string; } ' to 'IComplexType':
```

This compile error tells us that the `complexType_2` variable must conform to the `IComplexType` interface. The `complexType_2` variable has an `id` property, but it does not have a `name` property. To fix this error, and to ensure that the variable implements the `IComplexType` interface, we simply need to add a `name` property, as follows:

```
var complexType_2: IComplexType = {
    id: 2,
    name: "secondObject",
    description: "myDescription"
};
```

Even though we have an extra `description` property, the `IComplexType` interface only mentions the `id` and `name` properties – so as long as we have those, the object is said to be implementing the `IComplexType` interface.

Interfaces are a compile-time language feature of TypeScript, and the compiler does not generate any JavaScript code from interfaces that you include in your TypeScript projects. Interfaces are only used by the compiler for type checking during the compilation step.



In this book, we will be sticking to a simple naming convention for interfaces, and that is to prefix the interface name with the letter `I`. Using this naming scheme helps when dealing with large projects where code is spread across multiple files. Seeing anything prefixed with `I` in your code helps you distinguish it as an interface immediately. You can, however, call your interfaces anything.

Classes

A class is a definition of an object, what data it holds, and what operations it can perform. Classes and interfaces form a cornerstone of the principles of object-oriented programming, and often work together in design patterns. A design pattern is a simple programming structure that has been proven to be the best way of tackling a specific programming task. More on design patterns later.

Let's recreate our previous code sample using classes:

```
interface IComplexType {
    id: number;
    name: string;
    print(): string;
}
class ComplexType implements IComplexType {
    id: number;
    name: string;
    print(): string {
        return "id:" + this.id + " name:" + this.name;
    }
}

var complexType: ComplexType = new ComplexType();
complexType.id = 1;
complexType.name = "complexType";
```



```
var complexType_2: ComplexType = new ComplexType();
complexType_2.id = 2;
complexType_2.name = "complexType_2";

window.onload = () => {
    console.log(complexType.print());
    console.log(complexType_2.print());
}
```

Firstly, we have our interface definition (`IComplexType`), which has an `id` and a `name` property, as well as a `print` function. We then define a class named `ComplexType` that implements the `IComplexType` interface. In other words, the class definition for `ComplexType` must match the `IComplexType` interface definition. Note that the class definition does not create a variable – it simply defines the structure of the class. We then create a variable named `complexType`, and then assign to this variable a new instance of the `ComplexType` class. This line is said to be creating an instance of the class. Once we have an instance of the class, we can set the values of the class properties. The last section of the code simply calls the `print` function of each class inside a `window.onload` function. The output of this code is as follows:

```
id:1 name:complexType
id:2 name:complexType_2
```

Class constructors

Classes can accept parameters during their initial construction. If we look at the previous code sample, our calls to create an instance of a `ComplexType` class, and then set its properties, can be streamlined into a single line of code:

```
var complexType = new ComplexType(1, "complexType");
```

This version of the code is passing the `id` and `name` properties as parts of the class constructor. Our class definition, however, will need to include a new function, named `constructor`, in order to accept this syntax. Our updated class definition would then become:

```
class ComplexType implements IComplexType {
    id: number;
    name: string;
    constructor(idArg: number, nameArg: string) {
        this.id = idArg;
        this.name = nameArg;
    }
}
```

```
    print(): string {
        return "id:" + this.id + " name:" + this.name;
    }
}
```

Note the constructor function. It is a normal function definition, but uses the constructor keyword and accepts an `idArg`, and `nameArg` as parameters. These arguments are strongly typed to be of type `number` and `string` respectively. The internal `id` property of the `ComplexType` class is then assigned the `idArg` parameter value. Note the syntax used to reference the `id` property: `this.id`. Classes use the same `this` syntax that objects do to access internal properties. If we attempt to use an internal class property without using the `this` keyword, TypeScript will generate compile errors.

Class functions

All functions within a class adhere to the syntax and rules that we covered in the previous chapter on functions. As a refresher of these rules, all class functions can:

- Be strongly typed
- Use the `any` keyword to relax strong typing
- Have optional parameters
- Have default parameters
- Use argument arrays, or the rest parameter syntax
- Allow function callbacks and specify the function callback signature
- Allow function overloads

Let's modify our `ComplexType` class definition, and include an example of each of these rules:

```
class ComplexType implements IComplexType {
    id: number;
    name: string;
    constructor(idArg: number, nameArg: string);
    constructor(idArg: string, nameArg: string);
    constructor(idArg: any, nameArg: any) {
        this.id = idArg;
        this.name = nameArg;
    }
    print(): string {
        return "id:" + this.id + " name:" + this.name;
    }
}
```

```
    usingTheAnyKeyword(arg1: any): any {
        this.id = arg1;
    }
    usingOptionalParameters(optionalArg1?: number) {
        if (optionalArg1) {
            this.id = optionalArg1;
        }
    }
    usingDefaultParameters(defaultArg1: number = 0) {
        this.id = defaultArg1;
    }
    usingRestSyntax(...argArray: number []) {
        if (argArray.length > 0) {
            this.id = argArray[0];
        }
    }
    usingFunctionCallbacks( callback: (id: number) => string ) {
        callback(this.id);
    }
}
```

The first thing to note is the `constructor` function. Our class definition is using function overloading for the `constructor` function, allowing the class to be constructed using either a `number` and a `string`, or two `strings`. The following code shows how we would use each of these `constructor` definitions:

```
var complexType: ComplexType = new ComplexType(1, "complexType");
var complexType_2: ComplexType = new ComplexType("1", "1");
var complexType_3: ComplexType = new ComplexType(true, true);
```

The `complexType` variable uses the `number, string` variant of the `constructor` function, and the `complexType_2` variable uses the `string, string` variant. The `complexType_3` variable will generate a compile error, as we are not allowing a `constructor` to use a `boolean, boolean` variant. You may argue, however, that the last `constructor` function specifies an `any, any` variant, and this should allow for our `boolean, boolean` usage. Just remember that when using `constructor` overloads, the actual `constructor` implementation must use types that are compatible with any variant of the `constructor` overloads. Our `constructor` implementation, then, must use an `any, any` variant. Because we are using `constructor` overloads, however, this `any, any` variant is hidden by the compiler in favor of our overloaded signatures.

The following code samples show how we would use the rest of the functions that we have defined for this class. Let's start with the `usingTheAnyKeyword` function:

```
complexType.usingTheAnyKeyword(true);
complexType.usingTheAnyKeyword({id: 1, name: "test"});
```

The first call in this sample is using a boolean value to call the `usingTheAnyKeyword` function, and the second is using an arbitrary object. Both of these function calls are valid, as the parameter `arg1` is defined with the any type. Next, the `usingOptionalParameters` function:

```
complexType.usingOptionalParameters(1);
complexType.usingOptionalParameters();
```

Here, we are calling the `usingOptionalParameters` function firstly with a single argument, and then without any arguments. Again, these calls are valid, as the `optionalArg1` argument is marked as optional. Now for the `usingDefaultParameters` function:

```
complexType.usingDefaultParameters(2);
complexType.usingDefaultParameters();
```

Both of these calls to the `usingDefaultParameters` function are valid. The first call will override the default value of 0, and the second call—without an argument—will use the default value of 0. Next up is the `usingRestSyntax` function:

```
complexType.usingRestSyntax(1, 2, 3);
complexType.usingRestSyntax(1, 2, 3, 4, 5);
```

Our rest function, `usingRestSyntax`, can be called with any number of arguments, as we are using the rest parameter syntax to hold these arguments in an array. Both of these calls are valid. Finally, let's look at the `usingFunctionCallbacks` function:

```
function myCallbackFunction(id: number): string {
    return id.toString();
}
complexType.usingFunctionCallbacks(myCallbackFunction);
```

This snippet shows the definition of a function named `myCallbackFunction`. It matches the callback signature required by the `usingFunctionCallbacks` function, allowing us to pass in the `myCallbackFunction` as a parameter to the `usingFunctionCallbacks` function.

Note that if you face any difficulty understanding these various function signatures, then please re-view the relevant sections in *Chapter 2, Types, Variables, and Function Techniques*, regarding functions, where each of these concepts is explained in detail.

Interface function definitions

Interfaces, like classes, follow the same rules when dealing with functions. To update our `ComplexType` interface definition to match the `ComplexType` class definition, we need to write a function definition for each of the new functions, as follows:

```
interface IComplexType {
    id: number;
    name: string;
    print(): string;
    usingTheAnyKeyword(arg1: any): any;
    usingOptionalParameters(optionalArg1?: number);
    usingDefaultParameters(defaultArg1?: number);
    usingRestSyntax(...argArray: number []);
    usingFunctionCallbacks(callback: (id: number) => string);
}
```

Lines 1 to 4 form our existing interface definition, and include the `id` and `name` properties and the `print` function we have been using until now. Line 5 shows how to define a function signature for the `usingTheAnyKeyword` function. It looks surprisingly like our actual class function, but does not have a function body. Line 6 shows how to use an optional parameter for the `usingOptionalParameters` function. Line 7, however, is slightly different from our class definition of the `usingDefaultParameters` function. Remember that an interface defines the shape of our class or object, and therefore cannot contain variables or values. We have therefore defined the `defaultArg1` parameter as optional, and left the assignment of the default value up to the class implementation itself. Line 8 shows the definition of the `usingRestSyntax` function that contains the rest parameter syntax, and line 9 shows the definition of the `usingFunctionCallbacks` function, with a callback function signature. They are pretty much identical to the class function signatures.

The only thing missing from this interface is the signature for the `constructor` function. TypeScript will generate an error if we include a `constructor` signature in an interface. Suppose we were to include a definition for the `constructor` function in the `IComplexType` interface:

```
interface IComplexType {

    constructor(arg1: any, arg2: any);

}
```

The TypeScript compiler would then generate an error:

```
Types of property 'constructor' of types 'ComplexType' and 'IComplexType'
are incompatible
```

This error shows us that when we use a constructor function, the return type of the constructor is implicitly typed by the TypeScript compiler. Therefore, the return type of the `IComplexType` constructor would be `IComplexType`, and the return type of the `ComplexType` constructor would be `ComplexType`. Even though the `ComplexType` function implements the `IComplexType` interface, they are actually two different types – and therefore the constructor signatures will always be incompatible – hence the compile error.

Inheritance

Inheritance is another paradigm that is one of the cornerstones of object-oriented programming. Inheritance means that an object uses another object as its base type, thereby "inheriting" all of the base object's characteristics, including both properties and functions. Both interfaces and classes can use inheritance. An interface or class that is "inherited" from is known as the base interface, or base class, and the interface or class that does the inheritance is known as the derived interface, or derived class. TypeScript implements inheritance using the `extends` keyword.

Interface inheritance

As an example of interface inheritance, consider the following TypeScript code:

```
interface IBase {
    id: number;
}

interface IDerivedFromBase extends IBase {
    name: string;
}

class DerivedClass implements IDerivedFromBase {
    id: number;
    name: string;
}
```

We start with an interface called `IBase` that defines an `id` property, of type `number`. Our second interface definition, `IDerivedFromBase`, extends (or inherits) from `IBase`, and therefore automatically includes the `id` property. The `IDerivedFromBase` interface then defines a `name` property, of type `string`. As the `IDerivedFromBase` interface inherits from `IBase`, it therefore actually has two properties: `id` and `name`. The class definition for `DerivedClass` implements this `IDerivedFromBase` interface, and therefore must include both the `id` and `name` properties – in order to successfully implement all of the properties of the `IDerivedFromBase` interface. Although we have only shown properties in this example, the same rules apply for functions.

Class inheritance

Classes can also use inheritance in the same manner as interfaces. Using our definitions of the `IBase` and `IDerivedFromBase` interfaces, the following code shows an example of class inheritance:

```
class BaseClass implements IBase {
    id : number;
}

class DerivedFromBaseClass
    extends BaseClass
    implements IDerivedFromBase
{
    name: string;
}
```

The first class, named `BaseClass`, implements the `IBase` interface, and as such, is only required to define a property of `id`, of type `number`. The second class, `DerivedFromBaseClass`, inherits from the `BaseClass` class (using the `extends` keyword), but also implements the `IDerivedFromBase` interface. As `BaseClass` already defines the `id` property required in the `IDerivedFromBase` interface, the only other property that the `DerivedFromBaseClass` class needs to implement is the `name` property. We therefore only need to include the definition of the `name` property in the `DerivedFromBaseClass` class.

Function and constructor overloading with `super`

When using inheritance, it is often necessary to create a base class with a defined constructor. Then, in the constructor for any derived class, we will need to call through to the base class constructor and pass through these parameters. This is called constructor overloading. In other words, the constructor of a derived class overloads, or "supersedes", the constructor of the base class. TypeScript includes the `super` keyword to enable calling a base class's function with the same name. This is best explained with the following code snippet:

```
class BaseClassWithConstructor {
    private _id: number;
    constructor(id: number) {
        this._id = id;
    }
}

class DerivedClassWithConstructor extends BaseClassWithConstructor {
    private _name: string;
    constructor(id: number, name: string) {
        this._name = name;
        super(id);
    }
}
```

In this code snippet, we define a class named `BaseClassWithConstructor` that holds a private `_id` property. This class has a constructor function that requires an `id` argument. Our second class, named `DerivedClassWithConstructor`, inherits from, or extends, the `BaseClassWithConstructor` class. The constructor of `DerivedClassWithConstructor` takes an `id` argument and a `name` argument, but it needs to pass the `id` argument through to the base class. This is where the `super` call comes in. The `super` keyword calls the function in the base class that has the same name as the function in the derived class. The last line of the constructor function for `DerivedClassWithConstructor` shows the call using the `super` keyword, passing the `id` argument it received through to the base class constructor.

This technique is called function overloading. In other words, the derived class has a function name that is the same name as that of a base class function, and it "overloads" this function definition. We can use this technique on any function in a class— not only on constructors. Consider the following code snippet:

```
class BaseClassWithConstructor {
    private _id: number;
    constructor(id: number) {
        this._id = id;
    }
    getProperties(): string {
        return "_id:" + this._id;
    }
}

class DerivedClassWithConstructor extends BaseClassWithConstructor {
    private _name: string;
    constructor(id: number, name: string) {
        this._name = name;
        super(id);
    }
    getProperties(): string {
        return "_name:" + this._name + ", " + super.getProperties();
    }
}
```

The `BaseClassWithConstructor` class now has a function named `getProperties`, which just returns a string representation of the properties of the class. Our `DerivedClassWithConstructor` class, however, also includes a function called `getProperties`. This function is a function override of the `getProperties` base class function. In order to call through to the base class function, we need to include the `super` keyword, as shown in the call to `super.getProperties()`.

Here is an example usage of the preceding code:

```
window.onload = () => {
    var myDerivedClass = new DerivedClassWithConstructor(1, "name");
    console.log(
        myDerivedClass.getProperties()
    );
}
```

This code creates a variable named `myDerivedClass` and passes in the required arguments of `id` and `name`. We then simply log the result of the call to the `getProperties` function to the console. This code snippet will result in the following console output:

```
_name: name, _id: 1
```

The results show that the `getProperties` function of the `myDerivedClass` variable will call through to the base class `getProperties` function, as expected.

JavaScript closures

Before we continue with this chapter, let's take a quick look at how TypeScript implements classes in the generated JavaScript through a technique called closures. As we mentioned in *Chapter 1, TypeScript – Tools and Framework Options*, a closure is a function that refers to independent variables. These variables essentially remember the environment in which they were created. Consider the following JavaScript code:

```
function TestClosure(value) {
    this._value = value;
    function printValue() {
        console.log(this._value);
    }
    return printValue;
}

var myClosure = TestClosure(12);
myClosure();
```

Here, we have a function named `TestClosure` that takes a single parameter, named `value`. The body of the function first assigns the `value` argument to an internal property named `this._value`, and then defines an inner function named `printValue`, that logs the value of the `this._value` property to the console. The interesting bit is the last line in the `TestClosure` function – we are returning the `printValue` function.

Now take a look at the last two lines of the code snippet. We create a variable named `myClosure` and assign to it the result of calling the `TestClosure` function. Note that because we are returning the `printValue` function from inside the `TestClosure` function, this essentially also makes the `myClosure` variable a function. When we execute this function on the last line of the snippet, it will execute the inner `printValue` function, but remember the initial value of 12 that was used when creating the `myClosure` variable. The output of the last line of the code will log the value of 12 to the console.

This is the essential nature of closures. A closure is a special kind of object that combines a function with the initial environment in which it was created. In our preceding sample, since we stored whatever was passed in via the `value` argument into a local variable named `this._value`, JavaScript remembers the environment in which the closure was created, in other words, whatever was assigned to the `this._value` property at the time of creation will be remembered, and can be reused later.

With this in mind, let's take a look at the JavaScript that is generated by the TypeScript compiler for the `BaseClassWithConstructor` class we were just working with:

```
var BaseClassWithConstructor = (function () {
    function BaseClassWithConstructor(id) {
        this._id = id;
    }
    BaseClassWithConstructor.prototype.getProperties = function () {
        return "_id:" + this._id;
    };
    return BaseClassWithConstructor;
})();
```

Our closure starts with `function () {` on the first line, and ends with `}` on the last line. This closure first defines a function to be used as a constructor: `BaseClassWithConstructor(id)`. Bear in mind that when a JavaScript object is constructed, it inherits, or copies the `prototype` property of the original object into the new instance. In our sample, then, any object that is created using the `BaseClassWithConstructor` function will inherit the `getProperties` function as well – as it is part of the `prototype` property. Also, because the functions that are defined on the `prototype` property are also within the closure, they will remember the original execution environment, and variable values.

This closure is then surrounded with an opening bracket, `(`, on the first line, and a closing bracket, `)`, on the last line – defining what is known as a JavaScript function expression. This function expression is then immediately executed by the last two braces, `()`; . This technique of immediately executing a function is known as an **Immediately Invoked Function Expression (IIFE)**. Our IIFE above is then assigned to a variable named `BaseClassWithConstructor`, making it a first-class JavaScript object, and one that can be created with the `new` keyword. This is how TypeScript implements classes in JavaScript.

The implementation of the underlying JavaScript code that TypeScript uses for class definitions is actually a well-known JavaScript pattern – known as the **module** pattern. It uses closures to capture an execution environment, and also provides a way to expose a public API for classes, as seen by the use of the `prototype` property.

The good news is that an in-depth knowledge of closures, how to write them, and how to use the module pattern for defining classes – will all be taken care of by the TypeScript compiler – allowing us to focus on object-oriented principles without having to write JavaScript closures using this sort of boilerplate code.

The Factory Design Pattern

To illustrate how we can use interfaces and classes in a large TypeScript project, we will have a quick look at a very well-known object-oriented design pattern – the Factory Design Pattern.

Business requirements

As an example, let's assume that our business analyst gives us the following requirements:

You are required to categorize people, given their date of birth, and indicate with a `true` or `false` flag whether they are of a legal age to sign a contract. A person is deemed to be an infant if they are less than 2 years old. Infants cannot sign contracts. A person is deemed to be a child if they are less than 18 years old. Children cannot sign contracts either. A person is deemed to be an adult if they are more than 18 years of age, and only adults can sign contracts.

What the Factory Design Pattern does

The Factory Design Pattern uses a Factory class to return an instance of one of several possible classes based on the information provided to it.

The essence of this pattern is to place the decision-making logic for what type of class to create, in a separate class – the Factory class. The Factory class will then return one of several classes that are all subtle variations of each other, and which will do slightly different things based on their specialty. In order for our logic to work, any code that consumes one of these classes must have a common contract (or list of properties and methods) that all the variations of a class implement. This is the perfect scenario for an interface.

To implement our required business functionality, we will create an `Infant` class, a `Child` class, and an `Adult` class. The `Infant` and `Child` classes will return `false` when asked whether they can sign contracts, and the `Adult` class will return `true`.

The `IPerson` interface and the `Person` base class

According to our requirements, the class instance that is returned by the `Factory` must be able to do two things: print the category of the person in the required format, and tell us whether they can sign contracts or not. For completeness, we will include a third function that prints the date of birth. Let's define an interface to satisfy this requirement:

```
interface IPerson {
    getPersonCategory(): string;
    canSignContracts(): boolean;
    getDateOfBirth(): string;
}
```

Our `IPerson` interface has a `getPersonCategory` method that will return a string representation of their category: either "Infant", "Child", or "Adult". The `canSignContracts` method will return either `true` or `false`, and the `getDateOfBirth` method will simply return a printable version of their date of birth. To simplify our code, we will create a base class called `Person` that implements this interface, and will handle the common data and functions of all types of `Person`: storing and returning the date of birth. Our base class is defined as follows:

```
class Person {
    _dateOfBirth: Date
    constructor(dateOfBirth: Date) {
        this._dateOfBirth = dateOfBirth;
    }
    getDateOfBirth(): string {
        return this._dateOfBirth.toString();
    }
}
```

This `Person` class definition is the base class for each of our specialist types of person. As each one of our specialist classes will require a `getDateOfBirth` function, we can extract this common code into a base class. The constructor function requires a date, which is stored in the internal variable `_dateOfBirth`, and the `getDateOfBirth` function returns this `_dateOfBirth` converted into a string.

Specialist classes

Now for the three types of specialist classes:

```
class Infant extends Person implements IPerson {
    getPersonCategory(): string {
        return "Infant";
    }
    canSignContracts() { return false; }
}

class Child extends Person implements IPerson {
    getPersonCategory(): string {
        return "Child";
    }
    canSignContracts() { return false; }
}

class Adult extends Person implements IPerson
{
    getPersonCategory(): string {
        return "Adult";
    }
    canSignContracts() { return true; }
}
```

All of the classes in this snippet use inheritance to extend the `Person` class. Our `Infant`, `Child`, and `Adult` classes do not specify a constructor method, but instead inherit this constructor from their base class, `Person`. Each class implements the `IPerson` interface, and must therefore provide implementations of all three functions required by the `IPerson` interface definition. The `getDateOfBirth` function is defined in the `Person` base class, however, so each of these derived classes only needs to implement the `getPersonCategory` and `canSignContracts` functions to be valid. We can see that our `Infant` and `Child` classes return `false` for `canSignContracts`, and our `Adult` class returns `true`.

The Factory class

Now, let's move on to the `Factory` class itself. This class is responsible for holding all of the logic required to make decisions, and returns an instance of either an `Infant`, `Child`, or `Adult` class:

```
class PersonFactory {
    getPerson(dateOfBirth: Date): IPerson {
        var dateNow = new Date();
```

```
        var dateTwoYearsAgo = new Date(dateNow.getFullYear() - 2,
            dateNow.getMonth(), dateNow.getDay());
        var dateEighteenYearsAgo = new Date(dateNow.getFullYear() - 18,
            dateNow.getMonth(), dateNow.getDay());

        if (dateOfBirth >= dateTwoYearsAgo) {
            return new Infant(dateOfBirth);
        }
        if (dateOfBirth >= dateEighteenYearsAgo) {
            return new Child(dateOfBirth);
        }
        return new Adult(dateOfBirth);
    }
}
```

The `PersonFactory` class has only one function, `getPerson`, which returns an object of type `IPerson`. This function creates a variable named `dateNow`, that is set to the current date. This `dateNow` variable is then used to calculate two more variables, `dateTwoYearsAgo`, and `dateEighteenYearsAgo`. The decision logic then takes over, comparing the incoming `dateOfBirth` variable against these dates. This logic satisfies our requirements, and returns a new instance of either a new `Infant`, `Child`, or `Adult` class based on their date of birth.

Using the Factory class

To illustrate how to use this `PersonFactory` class, we will use the following code, wrapped in a `window.onload` function so that we can run it inside a browser:

```
window.onload = () => {
    var personFactory = new PersonFactory();

    var personArray: IPerson[] = new Array();
    personArray.push(personFactory.getPerson(
        new Date(2014, 09, 29))); // infant
    personArray.push(personFactory.getPerson(
        new Date(2000, 09, 29))); // child
    personArray.push(personFactory.getPerson(
        new Date(1950, 09, 29))); // adult

    for (var i = 0; i < personArray.length; i++) {
        console.log(" A person with a birth date of : "
            + personArray[i].getDateOfBirth()
            + " is categorised as : ")
    }
}
```

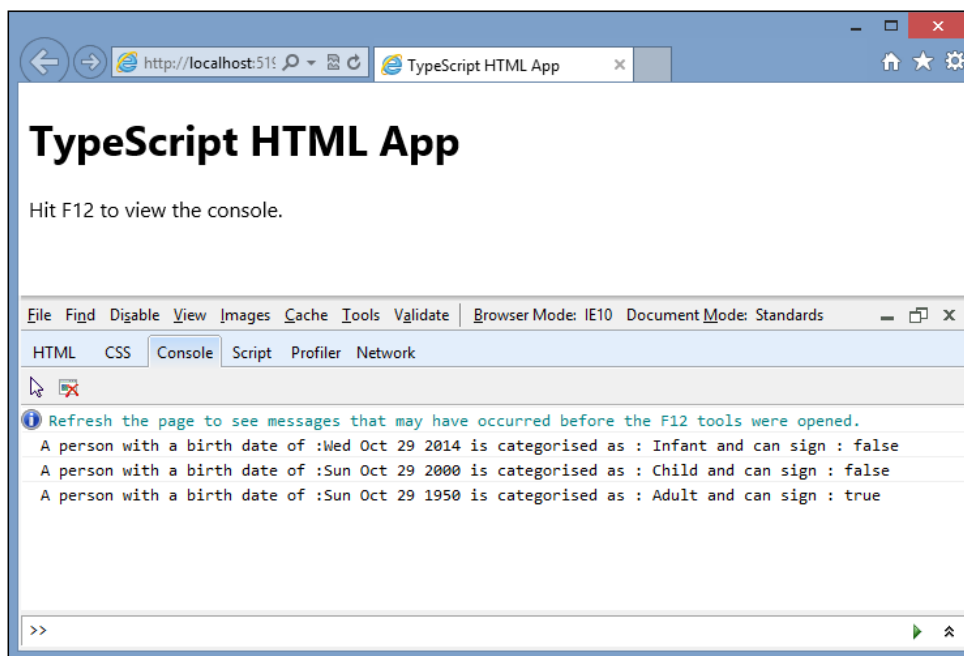
```

        + personArray[i].getPersonCategory()
        + " and can sign : "
        + personArray[i].canSignContracts());
    }
}

```

In line 2, we start with the creation of a variable, `personFactory`, to hold a new instance of the `PersonFactory` class. Line 4 creates a new array, named `personArray`, that is strongly typed to only hold objects that implement the `IPerson` interface. Lines 5 to 7 then add values to this array, by using the `getPerson` function of the `PersonFactory` class, passing in the date of birth. Note that the `PersonFactory` class will make all the decisions regarding which type of object to return, based on the date of birth we are passing in.

Line 8 starts a `for` loop to loop through the `personArray` array, and lines 9 to 14 use the interface definition of `IPerson` to call the relevant functions for printing. The output of this code is as follows:



We have satisfied our business requirements, and implemented a very common design pattern at the same time. If you find yourself repeating the same sort of logic in many places, trying to figure out whether an object falls under one or more categories, then chances are that you can refactor your code to use the Factory Design Pattern - and avoid repeating the same decision-making logic all over your code.

Class modifiers

As we discussed briefly in the opening chapter, TypeScript introduces the `public` and `private` access modifiers to mark variables and functions as either `public` or `private`. Traditionally, JavaScript programmers have used a simple naming convention of prefixing variables with an underscore (`_`) to indicate that they are private variables. This naming convention, however, does not stop anyone from actually modifying such variables inadvertently.

Let's take a look at a TypeScript code sample to illustrate this point:

```
class ClassWithModifiers {
  private _id: number;
  private _name: string;
  constructor(id: number, name: string) {
    this._id = id;
    this._name = name;
  }
  modifyId(id: number) {
    this._id = id;
    this.updateNameFromId();
  }
  private updateNameFromId() {
    this._name = this._id.toString() + "_name";
  }
}

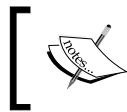
var myClass = new ClassWithModifiers(1, "name");
myClass.modifyId(2);
myClass._id = 2;
myClass.updateNameFromId();
```

We start with a class named `ClassWithModifiers`, which has two properties, `_id` and `_name`. We have marked these properties with the `private` keyword to protect them from being modified by mistake. Our `constructor` takes an incoming `id` and `name` parameter, and assigns the values to the internal, private properties of `_id` and `_name` respectively. The next function that we define is called `modifyId`, which will allow us to update the internal `_id` variable with a new value. The `modifyId` function then calls an internal function named `updateNameFromId`. This function has been marked as `private`, and therefore calls to it are only allowed within the body of the class definition. The `updateNameFromId` function simply uses the new `_id` value to set the private `_name` value.

The last four lines of code show us how we would use this class. The first line creates a variable named `myClass`, and assigns it to a new instance of the `ClassWithModifiers` class. The second line is legal, and calls the `modifyId` function. The third and fourth lines, however, will generate compile time errors:

```
error TS2107: Build: 'ClassWithModifiers._id' is inaccessible.
error TS2107: Build: 'ClassWithModifiers.updateNameFromId' is
inaccessible.
```

The TypeScript compiler warns us that both the `_id` property and `updateNameFromId` function are inaccessible – in other words, `private` – and are not designed to be used outside the class definition.



Class functions are `public` by default. Not specifying an access modifier of `private` for either properties or functions will cause their access level to default to `public`.

Constructor access modifiers


TypeScript also introduces a shorthand version of the previous constructor function, allowing you to specify parameters with access modifiers directly in the constructor. This is best described in code:

```
class ClassWithAutomaticProperties {
  constructor(public id: number, private name: string) {
  }
  print(): void {
    console.log("id:" + this.id + " name:" + this.name);
  }
}

var myAutoClass = new ClassWithAutomaticProperties(1, "name");
myAutoClass.id = 2;
myAutoClass.name = "test";
```

This code snippet defines a class named `ClassWithAutomaticProperties`. The constructor function uses two arguments - an `id` of type `number`, and a `name` of type `string`. Notice, however, the access modifiers of `public` for `id` and `private` for `name`. This shorthand automatically creates a `public id` property on the `ClassWithAutomaticProperties` class, and a `private name` property.


The `print` function on line 4 uses these automatic properties in the `console.log` function. We are referring to `this.id` and `this.name` in the `console.log` function, just as in our previous code samples.

 This shorthand syntax is available only within the constructor function.

We can see on line 9 that we have created a variable named `myAutoClass` and assigned a new instance of the `ClassWithAutomaticProperties` class to it. Once this class is instantiated, it automatically has two properties: an `id` property of type `number`, which is `public`; and a `name` property of type `string`, which is `private`. Compiling the previous code, however, will produce a TypeScript compile error:

```
error TS2107: Build: 'ClassWithAutomaticProperties.name' is inaccessible.
```

This error is telling us that the automatic property `name` is declared as `private`, and it is therefore unavailable to code outside the class itself.

 While this shorthand technique of creating automatic member variables is available, I believe that it makes the code more difficult to read. Personally, I prefer the more verbose class definitions that do not use this shorthand technique. With a list of properties at the top of the class, it becomes immediately visible to someone reading the code what variables this class uses, and whether they are `public` or `private`. Using the constructor's automatic property syntax hides these parameters somewhat, forcing developers to sometimes reread the code to understand it. Whichever syntax you choose, however, try to make it a coding standard, and use the same syntax throughout your code base.

Class property accessors

ECMAScript 5 introduces the concept of property accessors. This allows a pair of `get` and `set` functions (with the same function name) to be seen by the calling code as simple properties. This concept is best understood with some simple code samples:

```
class SimpleClass {
  public id: number;
}

var mySimpleClass = new SimpleClass();
mySimpleClass.id = 1;
```

Here, we have a class named `SimpleClass`, and it has a single public `id` property. When we create an instance of this class, we can directly modify this `id` property. Now let's use the ECMAScript 5 `get` and `set` functions to accomplish the same result:

```
class SimpleClassWithAccessors {
  private _id: number;
  get id() {
    return this._id;
  }
  set id(value: number) {
    this._id = value;
  }
}

var mySimpleAccClass = new SimpleClassWithAccessors();
mySimpleClass.id = 1;
console.log("id has the value of " + mySimpleClass.id);
```

This class has a private `_id` property and two functions, both called `id`. The first of these functions is prefixed by the `get` keyword and simply returns the value of the internal `_id` property. The second of these functions is prefixed with the `set` keyword and accepts a `value` parameter. The internal `_id` property is then set to this `value` parameter.

At the bottom of the class definition, we create a variable, named `mySimpleAccClass`, which is an instance of the `SimpleClassWithAccessors` class. Anyone using an instance of this class will not see two separate functions named `get` and `set`. They will simply see an `id` property. When we assign a value to this property, the ECMAScript 5 runtime will call the `set id(value)` function, and when we retrieve this property, the runtime will call the `get id()` function.



Some browsers do not support ECMAScript 5 (such as Internet Explorer 8), and will cause a JavaScript runtime error when this code is run.

Static functions

Static functions are functions that can be called on a class without having to create an instance of the class first. These functions are almost global in their nature, but must be called by prefixing the function name with the class name. Consider the following TypeScript code:

```
class ClassWithFunction {
  printOne() {
```

```
        console.log("1");
    }
}

var myClassWithFunction = new ClassWithFunction();
myClassWithFunction.printOne();
```

We start with a simple class, named `ClassWithFunction`, which has a single function, `printOne`. The `printOne` function does not really do anything useful, other than logging the string "1" to the console. In order to use this function, though, we need to first create an instance of the class, assign it to a variable, and then call the function.

With static functions, however, we can call functions or properties directly:

```
class StaticClass {
    static printTwo() {
        console.log("2");
    }
}

StaticClass.printTwo();
```

The class definition of `StaticClass` includes a single function, named `printTwo`, that is marked as `static`. As we can see from the last line of the code, we can call this function without "newing" up an instance of the `StaticClass` class. We can just call the function directly, as long as we prefix it with the class name.



Both functions and properties of a class can be marked as static.

Static properties

Static properties come in handy when dealing with so-called "magic strings" throughout your code base. If you are relying on a string to contain a particular value in various parts of your code, then the time has come to replace this "magic string" with a static property. In the Factory Design Pattern that we discussed earlier, we created specialist `Person` objects that returned either "Infant", "Child" or "Adult" as a string value. If we were writing code later on that checked whether the string returned was equal to "Infant" or "Child", we could inadvertently break our logic if we misspelled "Infant" as "Infent":

```
if (value === "Infant") {
    // do something with an infant.
}
```

The following is an example of static properties that we could use instead of those "magic strings":

```
class PersonType {
  static INFANT: string = "Infant";
  static CHILD: string = "Child";
  static ADULT: string = "Adult";
}
```

Then, in our code base, instead of checking values against the string "Infant", we compare them against the static property:

```
if (value === PersonType.INFANT) {
  // do something with an infant.
}
```

This code is not relying on a "magic string" anymore. The string "Infant" is now recorded in a single place. As long as all code uses the static property `PersonType.Infant`, it will be more stable and resistant to change.

Generics

Generics are a way of writing code that will deal with any type of object but still maintain the object type integrity. So far, we have used interfaces, classes and TypeScript's basic types to ensure strongly typed (and less error-prone) code in our samples. But what happens if a block of code needs to work with any type of object?

As an example, suppose we wanted to write some code that could iterate over an array of objects and return a concatenation of their values. So, given a list of numbers, say `[1, 2, 3]`, it should return the string `"1, 2, 3"`. Or, given a list of strings, say `["first", "second", "third"]`, return a string `"first, second, third"`. We could write some code that accepted values of type `any`, but this might introduce bugs in our code - remember S.F.I.A.T.? We want to ensure that all elements of the array are of the same type. This is where generics come in to play.

Generic syntax

Let's write a class called `Concatenator` that will work with any type of object, but still ensure that type integrity is kept in place. All JavaScript objects have a `toString` function, which is called whenever a string is needed by the runtime, so let's use this `toString` function to create a generic class that outputs all values held within an array.

A generic implementation of this Concatenator class is as follows:

```
class Concatenator< T > {
    concatenateArray(inputArray: Array< T >): string {
        var returnString = "";

        for (var i = 0; i < inputArray.length; i++) {
            if (i > 0)
                returnString += ",";
            returnString += inputArray[i].toString();
        }
        return returnString;
    }
}
```

The first thing we notice is the syntax of the class declaration, `Concatenator < T >`. This `< T >` syntax is the syntax used to indicate a generic type, and the name used for this generic type in the rest of our code is `T`. The `concatenateArray` function also uses this generic type syntax, `Array < T >`. This indicates that the `inputArray` argument must be an array of the type that was originally used to construct an instance of this class.

Instantiating generic classes

To use an instance of this generic class, we need to construct the class and tell the compiler via the `< >` syntax what the actual type of `T` is. We can use any type for the type of `T` in this generic syntax, including base JavaScript types, TypeScript classes, or even TypeScript interfaces:

```
var stringConcatenator = new Concatenator<string>();
var numberConcatenator = new Concatenator<number>();
var personConcatenator = new Concatenator<IPerson>();
```

Notice the syntax that we have used to instantiate the `Concatenator` class. In our first sample, we create an instance of the `Concatenator` generic class, and specify that it should substitute the generic type, `T`, with the type `string` in every place where `T` is being used within the code. Similarly, the second example creates an instance of the `Concatenator` class, and specifies that the type `number` should be used wherever the code encounters the generic type `T`. Our last sample shows the use of the `IPerson` interface for the generic type `T`.

If we use this simple substitution principle, then for the `stringConcatenator` instance (which uses strings), the `inputArray` argument must be of type `Array<string>`. Similarly, the `numberConcatenator` instance of this generic class uses numbers, and so the `inputArray` argument must be an array of numbers. To test this theory, let's generate an array of strings and an array of numbers, and see what the compiler says if we try to break this rule:

```
var stringArray: string[] = ["first", "second", "third"];
var numberArray: number[] = [1, 2, 3];
var stringResult = stringConcatenator.concatenateArray(stringArray);
var numberResult = numberConcatenator.concatenateArray(numberArray);
var stringResult2 = stringConcatenator.concatenateArray(numberArray);
var numberResult2 = numberConcatenator.concatenateArray(stringArray);
```

Our first two lines define our `stringArray` and `numberArray` variables to hold the relevant arrays. We then pass in the `stringArray` variable to the `stringConcatenator` function—no problems there. On our next line, we pass the `numberArray` to the `numberConcatenator`—still okay.

Our problems, however, start when we attempt to pass an array of numbers to the `stringConcatenator`, which has been configured to only use strings. Again, if we attempt to pass an array of strings to the `numberConcatenator`, which has been configured to allow only numbers, TypeScript will generate errors as follows:

```
Types of property 'pop' of types 'string[]' and 'number[]' are incompatible.
```

```
Types of property 'pop' of types 'number[]' and 'string[]' are incompatible.
```

The `pop` property is the first nonmatching property between a `string[]` and a `number[]`, so clearly, we are attempting to pass an array of numbers where we should have used strings, and vice versa. Again, the compiler warns us that we are not using the code correctly, and forces us to resolve these issues before continuing.



These constraints on generics are a compile-time-only feature of TypeScript. If we look at the generated JavaScript, we will not see any reams of code that jumps through hoops to ensure that these rules are carried through into the resultant JavaScript. All of these type constraints and generic syntax are simply compiled away. In the case of generics, the generated JavaScript is actually a very simplified version of our code, with no type constraints in sight.

Using the type T

When we use generics, it is important to note that all of the code within the definition of a generic class or a generic function must respect the properties of T as if it were any type of object. Let's take a closer look at the implementation of the `concatenateArray` function in this light:

```
class Concatenator< T > {
    concatenateArray(inputArray: Array< T >): string {
        var returnString = "";

        for (var i = 0; i < inputArray.length; i++) {
            if (i > 0)
                returnString += ",";
            returnString += inputArray[i].toString();
        }
        return returnString;
    }
}
```

The `concatenateArray` function strongly types the `inputArray` argument so that it should be of type `Array <T>`. This means that any code that uses the `inputArray` argument can use only those functions and properties that are common to all arrays, no matter what type the array holds. In this code sample, we used `inputArray` in two places.

Firstly, in our for loop, note where we have used the `inputArray.length` property. All arrays have a `length` property to indicate how many items the array has, so using `inputArray.length` will work on any array, no matter what type of object the array holds. Secondly, we reference an object within the array when we use the `inputArray[i]` syntax. This reference actually returns a single object of type T. Remember that whenever we use T in our code, we must use only those functions and properties that are common to any object of type T. Luckily for us, we are using only the `toString` function, and all JavaScript objects, no matter what type they are, have a valid `toString` function. So this generic code block will compile cleanly.

Lets test this type T theory by creating a class of our own to pass into the `Concatenator` class:

```
class MyClass {
    private _name: string;
    constructor(arg1: number) {
```

```
        this._name = arg1 + "_MyClass";
    }
}
var myArray: MyClass[] = [new MyClass(1), new MyClass(2),
    new MyClass(3)];
var myArrayConcatentator = new Concatenator<MyClass>();
var myArrayResult = myArrayConcatentator.concatenArray(myArray);
console.log(myArrayResult);
```

This sample starts with a class, named `MyClass`, that has a constructor accepting a number. It then assigns an internal variable called `_name` to a value of `arg1`, concatenated with the `"_MyClass"` string. Next, we create an array called `myArray`, and construct some instances of `MyClass` within this array. We then create an instance of the `Concatenator` class, specifying that this generic instance will only work with objects that are of type `MyClass`. We then call the `concatenArray` function and store the result in a variable named `myArrayResult`. Finally, we print the result on the console. Running this code in the browser will produce the following output:

```
[object Object],[object Object],[object Object]
```

Hmmm, not quite what we were expecting! This strange output is because the string representation of an object - that is not one of the basic JavaScript types - resolves to `[object type]`. Any custom object that you write may need to override the `toString` function to provide human-readable output. We can fix this code quite easily by providing an override of the `toString` function within our class, as follows:

```
class MyClass {
    private _name: string;
    constructor(arg1: number) {
        this._name = arg1 + "_MyClass";
    }
    toString(): string {
        return this._name;
    }
}
```

In the code above, we have replaced the default `toString` function that all JavaScript objects inherit, with our own implementation. Within this function, we simply returned the value of the `_name` private variable. Running this sample now produces the expected result:

```
1_MyClass,2_MyClass,3_MyClass
```

Constraining the type of T

When using generics, it is sometimes desirable to constrain the type of `T` to be only a specific type, or subset of types. In these cases, we don't want our generic code to be available for any type of object, we only want it to be available for a specific subset of objects. TypeScript uses inheritance to accomplish this with generics. As an example, let's refactor our earlier Factory Design Pattern code to use a generic `PersonPrinter` class, that is specifically designed to work with classes that implement the `IPerson` interface:

```
class PersonPrinter< T extends IPerson> {
    print(arg: T) {
        console.log("Person born on "
            + arg.getDateOfBirth()
            + " is a "
            + arg.getPersonCategory()
            + " and is " +
            this.getPermissionString(arg)
            + "allowed to sign."
        );
    }
    getPermissionString(arg: T) {
        if (arg.canSignContracts())
            return "";
        return "NOT ";
    }
}
```

In this code snippet, we define a class called `PersonPrinter`, that uses the generic syntax. Note that the `T` generic type has been derived from the `IPerson` interface, as indicated by the `extends` keyword in `< T extends IPerson >`. This indicates that any usage of the type `T` will substitute the interface `IPerson`, and can therefore, only allow functions or properties that are defined in the `IPerson` interface to be used wherever `T` is used. The `print` function accepts an argument named `arg`, which is of type `T`. Using our rules of generics, we know that any usage of the variable `arg` is only allowed to use available functions from the `IPerson` interface.

The `print` function builds up a string to log to the console, and only uses functions that are defined in the `IPerson` interface. These include the functions `getDateOfBirth` and `getPersonCategory`. In order to generate a grammatically correct sentence, we have introduced another function called `getPermissionString` that accepts an argument of type `T`, or the `IPerson` interface. This function simply uses the `canSignContracts()` function of the `IPerson` interface to return either a blank string or the string "NOT ".

To illustrate the usage of this class, consider the following code:

```
window.onload = () => {
    var personFactory = new PersonFactory();
    var personPrinter = new PersonPrinter<IPerson>();

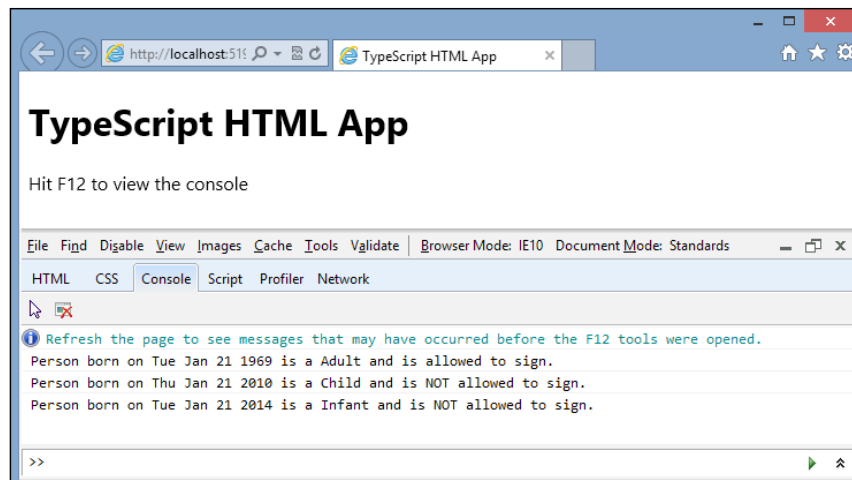
    var child = personFactory.getPerson(new Date(2010, 0, 21));
    var adult = personFactory.getPerson(new Date(1969, 0, 21));
    var infant = personFactory.getPerson(new Date(2014, 0, 21));

    console.log(personPrinter.print(adult));
    console.log(personPrinter.print(child));
    console.log(personPrinter.print(infant));
}
```

First, we create a new instance of the `PersonFactory` class. We then create an instance of the generic `PersonPrinter` class, and set the type of the argument `T` to be of type `IPerson`. This means that any class that is passed into the instance of `PersonPrinter` must implement the `IPerson` interface. We know from our previous examples that the `PersonFactory` will return an instance of either an `Infant`, `Child`, or `Adult` class, and each of these classes implement the `IPerson` interface. We know therefore, that any class returned by the `PersonFactory` will be accepted by the `personPrinter` generic class instance.

Next, we instantiate variables named `child`, `adult`, and `infant`, and rely on the `PersonFactory` to return us the correct class based on their date of birth. The last three lines of this sample simply log to the console the sentence that is generated by the `personPrinter` generic class instance.

The output of this code is as we expected:



Generics PersonFactory output

Generic interfaces

We can also use interfaces with the generic type syntax. For our `PersonPrinter` class, the matching interface definition would be:

```
interface IPersonPrinter<T extends IPerson> {
    print(arg: T) : void;
    getPermissionString(arg: T): string;
}
```

This interface looks identical to our class definition, the only difference being that the `print` and the `getPermissionString` functions do not have an implementation. We have kept the generic type syntax using `< T >`, and further specified that the type `T` must implement the `IPerson` interface. To use this interface with the `PersonPrinter` class, we modify the class definition, as follows:

```
class PersonPrinter<T extends IPerson> implements IPersonPrinter<T> {
}
```

This syntax seems pretty straightforward. As we have seen before, we use the `implements` keyword following the class definition, and then use the interface name. Note, however, that we pass the type `T` into the interface definition of `IPersonPrinter` as a generic type `IPersonPrinter<T>`. This satisfies the `IPersonPrinter` generic interface definition.

An interface that defines our generic classes further protects our code from being modified inadvertently. As an example of this, suppose that we tried to redefine the class definition of `PersonPrinter` so that `T` is not constrained to be of type `IPerson`:

```
class PersonPrinter<T> implements IPersonPrinter<T> {
}
```

Here, we have removed the constraint on the type `T` for the `PersonPrinter` class. TypeScript will automatically generate an error:

```
Type 'T' does not satisfy the constraint 'IPerson' for type parameter 'T extends IPerson'.
```

This error points us to our erroneous class definition; the type `T`, as used in the code (`PersonPrinter<T>`), must use a type `T` that extends from `IPerson`.

Creating new objects within generics

From time to time, generic classes may need to create an object of the type that is passed in as the generic type `T`. Consider the following code:

```
class FirstClass {
    id: number;
}

class SecondClass {
    name: string;
}

class GenericCreator< T > {
    create(): T {
        return new T();
    }
}

var creator1 = new GenericCreator<FirstClass>();
var firstClass: FirstClass = creator1.create();

var creator2 = new GenericCreator<SecondClass>();
var secondClass : SecondClass = creator2.create();
```

Here, we have two class definitions, `FirstClass` and `SecondClass`. `FirstClass` just has a public `id` property, and `SecondClass` has a public `name` property. We then have a generic class that accepts a type `T` and has a single function, named `create`. This `create` function attempts to create a new instance of the type `T`.

The last four lines of the sample show us how we would like to use this generic class. The `creator1` variable creates a new instance of the `GenericCreator` class using the correct syntax for creating variables of type `FirstClass`. The `creator2` variable is a new instance of the `GenericCreator` class, but this time is using `SecondClass`. Unfortunately, the preceding code will generate a TypeScript compile error:

```
error TS2095: Build: Could not find symbol 'T'.
```

According to the TypeScript documentation, in order to enable a generic class to create objects of type `T`, we need to refer to type `T` by its constructor function. We also need to pass in the class definition as an argument. The `create` function will need to be rewritten as follows:

```
class GenericCreator< T > {
    create(arg1: { new(): T }) : T {
        return new arg1();
    }
}
```

Let's break this `create` function down into its component parts. First, we pass an argument, named `arg1`. This argument is then defined to be of type `{ new(): T }`. This is the little trick that allows us to refer to `T` by its `constructor` function. We are defining a new anonymous type that overloads the `new()` function and returns a type `T`. This means that the `arg1` argument is a function that is strongly typed to have a single `constructor` that returns a type `T`. The implementation of this function simply returns a new instance of the `arg1` variable. Using this syntax removes the compile error that we encountered before.

This change, however, means that we must pass the class definition to the `create` function, as follows:

```
var creator1 = new GenericCreator<FirstClass>();
var firstClass: FirstClass = creator1.create(FirstClass);

var creator2 = new GenericCreator<SecondClass>();
var secondClass : SecondClass = creator2.create(SecondClass);
```

Note the change in usage of the `create` function on lines 2 and 5. We are now required to pass in the class definition for our type of `T`: `create(FirstClass)` and `create(SecondClass)` as our first argument. Try running this code in your browser and see what happens. The generic class will, in fact, create new objects of types `FirstClass` and `SecondClass`, as we expected.

Runtime type checking

Although the TypeScript compiler generates compilation errors for incorrectly typed code, this type checking is compiled away in the generated JavaScript. This means that the JavaScript runtime engine knows nothing about TypeScript interfaces or generics. So how can we tell at runtime whether a class implements an interface?

JavaScript has some functions that we can use when dealing with objects, that will tell us what type an object is, or if one object is an instance of another. For type information, we can use the JavaScript `typeof` keyword, and for instance information, we can use `instanceof`. Let's have a look at what these functions return, given some simple TypeScript classes, and see if we can use these to tell whether a class implements an interface.

First, a simple base class:

```
class TcBaseClass {
  id: number;
  constructor(idArg: number) {
    this.id = idArg;
  }
}
```

This `TcBaseClass` class has an `id` property, and a constructor that sets this property based on the argument passed to it.

Then, a class that is derived from `TcBaseClass`:

```
class TcDerivedClass extends TcBaseClass {
  name: string;
  constructor(idArg: number, nameArg: string) {
    super(idArg);
    this.name = name;
  }
  print() {
    console.log(this.id + " " + this.name);
  }
}
```

This `TcDerivedClass` class derives (or extends) from the `TcBase` class, and adds a `name` property and a `print` function. The constructor of this derived class must call the constructor of the base class, passing in the `idArg` argument via the `super` function.

Now, let's construct a variable named `base` that is a new instance of `TcBaseClass`, and then construct a variable named `derived` that is a new instance of `TcDerivedClass`, as follows:

```
var base = new TcBaseClass(1);
var derived = new TcDerivedClass(2, "second");
```

Now for some tests; let's see what the `typeof` function returns for each of these classes:

```
console.log("typeof base: " + typeof base);
console.log("typeof derived: " + typeof derived);
```

This code will return:

```
typeof base: object
typeof derived: object
```


This tells us that the JavaScript runtime engine sees an instance of a class as an object.

Let's now switch over to the `instanceof` keyword, and use it to check whether an object is derived from another:

```
console.log("base instance of TcBaseClass : " +
    (base instanceof TcBaseClass));
console.log("derived instance of TcBaseClass: " +
    (derived instanceof TcBaseClass));
```

This code will return:

```
base instance of TcBaseClass : true
derived instance of TcBaseClass: true
```

So far so good. Now let's have a look at what the `typeof` keyword returns when we use it on a class's properties:

```
console.log("typeof base.id: " + typeof base.id);
console.log("typeof derived.name: " + typeof derived.name);
console.log("typeof derived.print: " + typeof derived.print);
```

This code will return:

```
typeof base.id: number
typeof derived.name: string
typeof derived.print: function
```

As we can see, the JavaScript runtime correctly identifies the `id` property of our base type as a number, the `name` property as a string, and the `print` property as a function.

So how can we tell at runtime what the type of an object is? The simple answer is that we can't easily tell. We can only tell whether an object is an instance of another object, or if a property is one of the basic JavaScript types. If we were trying to use the `instanceof` function to implement a type checking algorithm, we would need to check the incoming object against every known type in our object tree, which is certainly not ideal. We also can't use `instanceof` to check whether a class implements an interface, as TypeScript interfaces are compiled away.

Reflection

Other statically typed languages allow the runtime engine to query an object, determine what type the object is, and also query what interfaces an object implements. This process is called reflection.

As we have seen, using the `typeof` or `instanceof` JavaScript functions, we can glean some information from the runtime about an object. On top of these abilities, we could also use the `getPrototypeOf` function to return some information about the class constructor. The `getPrototypeOf` function returns a string, so we could then parse this string to determine the class name. Unfortunately, the implementation of the `getPrototypeOf` function returns slightly different strings, depending on what browser is being used. It is also only implemented in ECMAScript 5.1 and above, which again, may introduce problems when running on older browsers or mobile browsers.

Another JavaScript function we could use to find runtime information about an object is the `hasOwnProperty` function. This has been a part of JavaScript since ECMAScript 3, and so is compatible with just about every browser, both desktop and mobile. The `hasOwnProperty` function will return `true` or `false`, indicating whether an object has the property that you are looking for.

The TypeScript compiler helps us program JavaScript in an object-oriented way using interfaces, but these interfaces are "compiled away", and do not appear in the generated JavaScript. As an example of this, let's have a look at the following TypeScript code:

```
interface IBasicObject {
    id: number;
    name: string;
    print(): void;
}

class BasicObject implements IBasicObject {
    id: number;
    name: string;
    constructor(idArg: number, nameArg: string) {
        this.id = idArg;
        this.name = nameArg;
    }
    print() {
        console.log("id:" + this.id + ", name" + this.name);
    }
}
```

This is a simple example of defining an interface and implementing it in a class. The `IBasicObject` interface has an `id` of type `number`, a `name` of type `string`, and a `print` function. The class definition `BasicObject` implements all the required properties and parameters. Now let's have a look at the compiled JavaScript that TypeScript generates:

```
var BasicObject = (function () {
    function BasicObject(idArg, nameArg) {
        this.id = idArg;
        this.name = nameArg;
    }
    BasicObject.prototype.print = function () {
        console.log("id:" + this.id + ", name" + this.name);
    };
    return BasicObject;
})();
```

The TypeScript compiler has not included any JavaScript for the `IBasicObject` interface. All we have here is a closure pattern for the `BasicObject` class definition. The `IBasicObject` interface, although used by the TypeScript compiler, does not exist in the generated JavaScript. Hence, we say that it has been "compiled away".

This therefore presents us with a few problems when implementing reflection-like capabilities within JavaScript:

- We cannot tell at runtime whether an object implements a TypeScript interface because TypeScript interfaces are compiled away
- We cannot loop through an object's properties using the `getOwnPropertyNames` function on older ECMAScript 3 browsers
- We cannot use the `getPrototypeOf` function on older ECMAScript 3 browsers to determine a class name
- The implementation of the `getPrototypeOf` function is not consistent across browsers
- We cannot use the `instanceof` keyword to determine a class type without comparing it with known types

Checking an object for a function

So how do we tell at runtime whether an object implements an interface?

In their book, *Pro JavaScript Design Patterns* (<http://jsdesignpatterns.com/>), Ross Harmes and Dustin Diaz discuss this quandary, and come up with a rather simple solution. We can invoke a function on an object using a string which contains the function name, and then check whether the result is valid, or undefined. In their book, they build a utility function using this principle, to check at runtime whether an object has a set of defined properties and methods. These defined properties and methods are kept within the JavaScript code as simple string arrays. These string arrays therefore act as object "metadata" for our code that we can then pass through to a function checking utility.

Their `FunctionChecker` utility class can be written in TypeScript as follows:

```
class FunctionChecker {
    static implementsFunction(
        objectToCheck: any, functionName: string): boolean
    {
        return (objectToCheck[functionName] != undefined &&
            typeof objectToCheck[functionName] == 'function');
    }
}
```

This `FunctionChecker` class has a single static function, named `implementsFunction`, that will return either `true` or `false`. The `implementsFunction` function takes an argument named `objectToCheck` and a string named `functionName`. Note that the type of `objectToCheck` is specifically set to `any`. This is one of the rare circumstances where the use of the `any` type is actually the correct TypeScript type.

Within the `implementsFunction` function, we use a special kind of JavaScript syntax that reads the function itself from the object, using the `[]` syntax on an instance of the object, and referencing it by name: `objectToCheck[functionName]`. If the object we are interrogating has this attribute, then invoking it will return something other than `undefined`. We can then use the `typeof` keyword to check the type of the attribute. If the `typeof` instance returns "function", then we know that this object implements this function. Let's have a look at some quick usages:

```
var myClass = new BasicObject(1, "name");
var isValidFunction = FunctionChecker.implementsFunction(
    myClass, "print");
console.log("myClass implements the print() function : "
    + isValidFunction);
```

```
isValidFunction = FunctionChecker.implementsFunction(  
    myClass, "alert");  
console.log("myClass implements the alert() function :"  
    + isValidFunction);
```

Line 1, simply creates an instance of the `BasicObject` class, and assigns it to the `myClass` variable. Line 2 then invokes our `implementsFunction` function, passing in the instance of the class and the string "print". Line 3 logs the result to the console. Line 4 and 5 repeat the process, but check whether the `myClass` instance implements the function "alert". The results of this code would be the following:

```
myClass implements the print() function :true  
myClass implements the alert() function :false
```

This `implementsFunction` function allows us to interrogate an object and check whether it has a specific function by name. Extending this concept slightly, brings us to a simple way of carrying out runtime type checking. All we need is a list of functions (or properties) that a JavaScript object should implement. This list of functions (or properties) can be described as class "metadata".

Interface checking with generics

This technique that Ross and Dustin describe, of holding "metadata" information about interfaces, is easily implemented in TypeScript. If we define classes that hold this "metadata" for each of our interfaces, we can then use them to check objects at runtime. Let's put together an interface that holds an array of method names to check an object against, as well as a list of property names.

```
interface IInterfaceChecker {  
    methodNames?: string[];  
    propertyNames?: string[];  
}
```

This `IInterfaceChecker` interface is very simple – an optional array of `methodNames`, and an optional array of `propertyNames`. Now let's implement this interface to describe the necessary properties and methods of the TypeScript `IBasicObject` interface:

```
class IBasicObject implements IInterfaceChecker {  
    methodNames: string[] = ["print"];  
    propertyNames: string[] = ["id", "name"];  
}
```

We start off with a class definition that implements the `IInterfaceChecker` interface. This class has been named `IIBasicObject`, with a double `I` prefix in the class name. This is a simple naming convention that indicates that the `IIBasicObject` class holds "metadata" for the `IBasicObject` interface that we defined earlier. The `methodNames` array specifies that this interface must implement the `print` method, and the `propertyNames` array specifies that this interface also includes an `id` and a `name` property.

This method of defining metadata for an object is a very simple solution to our problem, and is both browser agnostic and ECMAScript version agnostic. While this may require us to keep "metadata" objects in sync with TypeScript interfaces, we now have what we need in order to check whether an object implements a defined interface.

We can also use what we know about generics to implement an `InterfaceChecker` class that uses these object "metadata" classes:

```
class InterfaceChecker<T extends IInterfaceChecker> {
  implementsInterface(
    classToCheck: any,
    t: { new (): T; }
  ): boolean
  {
    var targetInterface = new t();
    var i, len: number;
    for (i = 0, len = targetInterface.methodNames.length;
        i < len; i++) {
      var method: string = targetInterface.methodNames[i];
      if (!classToCheck[method] ||
          typeof classToCheck[method] !== 'function') {
        console.log("Function :" + method + " not found");
        return false;
      }
    }
    for (i = 0, len = targetInterface.propertyNames.length;
        i < len; i++) {
      var property: string = targetInterface.propertyNames[i];
      if (!classToCheck[property] ||
          typeof classToCheck[property] !== 'function') {
        console.log("Property :" + property + " not found");
        return false;
      }
    }
  }
}
```

```
        return true;
    }
}
var myClass = new BasicObject(1, "name");
var interfaceChecker = new InterfaceChecker();

var isValid = interfaceChecker.implementsInterface(myClass,
    IIBasicObject);

console.log("myClass implements the IIBasicObject interface :"+
    + isValid);
```

We start off with a generic class, named `InterfaceChecker`, that accepts any object `T` that implements the `IInterfaceChecker` class. Again, the definition of the `IInterface` class is just an array of `methodNames` and an array of `propertyNames`. This class only has a single function named `implementsInterface` that returns a boolean – true if the class implements all properties and methods, and false if it does not. The first parameter, `classToCheck`, is the class instance that we are interrogating against the interface "metadata". Our second parameter uses the generic syntax that we discussed earlier to be able to create a new instance of the type `T` – which in this case is any type that implements the `IInterfaceChecker` interface.

The body of the code is an extension of the `FunctionChecker` class that we discussed earlier. We first need to create an instance of the type `T`, which is assigned to the variable `targetInterface`. We then simply loop through all the strings in the `methodNames` array, and check whether our `classToCheck` object implements these functions.

We then repeat this process, checking the given strings in the `propertyNames` array.

The last lines of this code sample show us how we would use this `InterfaceChecker` class. First, we create an instance of `BasicObject` and assign it to the variable `myClass`. We then create an instance of the `InterfaceChecker` class and assign it to the variable `interfaceChecker`.

The second last line of this snippet calls the `implementsInterface` function, passing in the `myClass` instance, and `IIBasicObject`. Note that we are not passing in an instance of the `IIBasicObject` class, we are just passing in the class definition. Our generic code will create an internal instance of the `IIBasicObject` class.

The last line of this code simply logs a `true` or `false` message to the console. The output of this line would be:

```
myClass implements the IIBasicObject interface :true
```

Let's now run the code with an invalid object:

```
var noPrintFunction = { id: 1, name: "name" };
isValid = interfaceChecker.implementsInterface(
    noPrintFunction, IIBasicObject);
console.log("noPrintFunction implements the
    IIBasicObject interface:" + isValid);
```

The variable `noPrintFunction` has both an `id` and a `name` property, but it does not implement a `print` function. The output of this code would be:

```
Function :print not found
noPrintFunction implements the IIBasicObject interface :false
```

We now have a way of determining at runtime whether or not an object implements a defined interface. This technique can be used on external JavaScript libraries that you do not control—or even in larger teams where the API for a particular library is agreed in principle, before the libraries are written. In these cases, once a new version of the library is delivered, the consumers can quickly and easily ensure that the API conforms to the design specification.

Interfaces are used in a number of design patterns, and even though we can implement these patterns using TypeScript, we may want to further solidify our code by doing runtime checking of an object's interface. This technique also opens up the possibility of writing an **Inversion of Control (IOC)** container in TypeScript, or an implementation of the Domain Events Pattern. We will explore these two design patterns in more detail in *Chapter 8, Object-oriented Programming with TypeScript*.

Summary

In this chapter, we explored the object-oriented concepts of interfaces, classes and generics. We discussed both interface inheritance and class inheritance, and used our knowledge on interfaces, classes and inheritance to create a Factory Design Pattern implementation in TypeScript. We then moved on to generics and their syntax, generic interfaces and generic constructor functions. We finished the chapter off with a discussion on reflection, and implemented a TypeScript version of an `InterfaceChecker` pattern using generics. In the next chapter, we will look at the mechanism that TypeScript uses to integrate with existing JavaScript libraries—definition files.

4

Writing and Using Declaration Files

One of the most appealing facets of JavaScript development is the myriad of external JavaScript libraries that have already been published, such as jQuery, Knockout, and Underscore. The TypeScript designers knew that introducing "syntactic sugar" to the TypeScript language would bring a range of benefits to the developer experience. These benefits include IDE features such as Intellisense, as well as detailed compile time error messages. We have already seen how to use this syntax for most of the TypeScript language features such as classes, interfaces, and generics, but how do we apply this "sugar" to existing JavaScript libraries? The answer is relatively simple – declaration files.

A declaration file is a special type of file used by the TypeScript compiler. It is marked with a `.d.ts` extension, and is then used by the TypeScript compiler within the compilation step. Declaration files are similar to header files used in other languages; they simply describe the syntax and structure of available functions and properties, but do not provide an implementation. Declaration files, therefore, do not actually generate any JavaScript code. They are there simply used to provide TypeScript compatibility with external libraries, or to fill in the gaps for JavaScript code that TypeScript does not know about. In order to use any external JavaScript library within TypeScript, you will need a declaration file.

In this chapter, we will explore declaration files, show the reasoning behind them, and build one based on some existing JavaScript code. If you are familiar with declaration files and how to use them, then you may be interested in the *Declaration Syntax Reference* section. This section is designed as a quick reference guide to the module definition syntax. Since writing declaration files is a rather small part of TypeScript development, we do not write them very often. The *Declaration Syntax Reference* section shows sample declaration file syntax for the equivalent JavaScript syntax.

Global variables

Most modern websites use some sort of server engine to generate the HTML for their web pages. If you are familiar with the Microsoft stack of technologies, then you would know that ASP.NET MVC is a very popular server-side engine, used to generate HTML pages based on master pages, partial pages, and MVC views. If you are a Node developer, then you may be using one of the popular Node packages to help you construct web pages through templates, such as Jade or Embedded JavaScript (EJS).

Within these templating engines, you may sometimes need to set JavaScript properties on the HTML page as a result of your backend logic. As an example, let's assume that you keep a list of contact e-mail addresses on your backend database, and then surface these to your frontend HTML page through a JavaScript global variable named `CONTACT_EMAIL_ARRAY`. Your rendered HTML page would then include a `<script>` tag that contains this global variable and contact e-mail addresses. You may have some JavaScript that reads this array, and then renders the values in a footer. The following HTML sample shows what a generated script within an HTML page may end up looking like:

```
<body>
  <script type="text/javascript">
    var CONTACT_EMAIL_ARRAY = [
      "help@site.com",
      "contactus@site.com",
      "webmaster@site.com"
    ];
  </script>
</body>
```

This HTML has a script block and within this script block, some JavaScript. The JavaScript is simply a variable named `CONTACT_EMAIL_ARRAY` that contains some strings. Let's assume that we wanted to write some TypeScript that can read this global variable. Consider the following TypeScript code:

```
class GlobalLogger {
  static logGlobalsToConsole() {
    for (var i = 0; i < CONTACT_EMAIL_ARRAY.length; i++) {
      console.log("found contact : " + CONTACT_EMAIL_ARRAY[i]);
    }
  }
}

window.onload = () => {
  GlobalLogger.logGlobalsToConsole();
}
```

This code creates a class named `GlobalLogger` with a single static function named `logGlobalsToConsole`. The function simply iterates through the `CONTACT_EMAIL_ARRAY` global variable, and logs the items in the array to the console.

If we compile this TypeScript code, we will generate the following errors:

```
error TS2095: Build: Could not find symbol 'CONTACT_EMAIL_ARRAY'.
```

This error indicates that the TypeScript compiler does not know anything about the variable named `CONTACT_EMAIL_ARRAY`. It does not even know that it is an array. As this piece of JavaScript is outside any TypeScript code, we will need to treat it in the same way as "external" JavaScript.

To solve our compilation problem, and make this `CONTACT_EMAIL_ARRAY` variable visible to TypeScript, we will need to use a declaration file. Let's create a file named `globals.d.ts` and include the following TypeScript declaration within it:

```
declare var CONTACT_EMAIL_ARRAY: string [];
```

The first thing to notice is that we are using a new TypeScript keyword: `declare`. The `declare` keyword tells the TypeScript compiler that we want to define the type of something, but that the implementation of this object (or variable or function) will be resolved at runtime. We have declared a variable named `CONTACT_EMAIL_ARRAY` that is typed to be an array of strings. This `declare` keyword does two things for us: it allows the use of the variable `CONTACT_EMAIL_ARRAY` within TypeScript code, and it also strongly types this variable to be an array of strings.



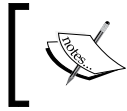
The 1.0 version and upwards of the TypeScript compiler will scan our source code directory for `.d.ts` files and automatically include them in the compilation step. In previous versions, it was necessary to include a comment as a reference to these files, but this reference comment line is no longer necessary.

With this `globals.d.ts` file in place, our code compiles correctly. If we now run this in a browser, the output will be as follows:

```
found contact : help@site.com
found contact : contactus@site.com
found contact : webmaster@site.com
```

So, by using a declaration file named `globals.d.ts`, we have been able to describe the structure of an "external" JavaScript variable to the TypeScript compiler. This JavaScript variable is defined outside any of our TypeScript code, yet we are still able to work with the definition of this variable within TypeScript.

This is what declaration files are used for. We are basically telling the TypeScript compiler to use the definitions found within a declaration file within the compilation step, and that the actual variables themselves will only be available at runtime.



Definition files also bring Intellisense or code completion functionality to our IDE for external JavaScript libraries and code.

Using JavaScript code blocks in HTML

The samples we have just seen are an example of tight coupling between the generated HTML content (that contains JavaScript code in script blocks) on your web page, and the actual running JavaScript. You may argue, however, that this is a design flaw. If the web page needed an array of contact e-mails, then the JavaScript application should simply send an AJAX request to the server for the same information in JSON format. While this is a very valid argument, there are cases where including content in the rendered HTML is actually faster.

There used to be a time where the Internet seemed to be capable of sending and receiving vast amounts of information in the blink of an eye. Bandwidth and speed on the Internet were growing exponentially, and desktops were getting larger amounts of RAM and faster and faster processors. As developers during this stage of the Internet highway, we stopped thinking about how much RAM a typical user had on their machine. We also stopped thinking about how much data we were sending across the wire. This was because Internet speeds were so fast and browser processing speed was seemingly limitless.

Yeah, and then along came the mobile phone -and it felt like we were back in the 1990s -with incredibly slow Internet connections, tiny screen resolutions, limited processing power, very little RAM (and popular arcade gaming experiences like *Elevator Action*, found at https://archive.org/details/Elevator_Action_1985_Sega_Taito_JP_en). The point of this story is that as modern web developers, we still need to be mindful of browsers that run on mobile phones. These browsers are sometimes running on very limited Internet connections, meaning that we must carefully measure the size of our JavaScript libraries, JSON data, and HTML pages, to ensure that our applications are fast and usable, even on mobile browsers.

This technique of including JavaScript variables or smaller static JSON data within the rendered HTML page often provides us with the fastest way to render a screen on an older browser, or in the modern age, a mobile phone. Many popular sites use this technique to quickly render the general structure of the page (header, side panels, footers, and so on) before the main content is delivered through asynchronous JSON requests. This technique works well because it renders the page faster and gives the user faster feedback.

Structured data

Let's enhance this simple array of contact e-mails with a little more relevant data. For each of these e-mail addresses, we now want to include some text that we will render within the footer of our page, along with the e-mail addresses. Consider the following HTML code with a global variable that uses a JSON structure:

```
<script type="text/javascript">
  var CONTACT_DATA = [
    { DisplayText: "Help", Email: "help@site.com" },
    { DisplayText: "Contact Us", Email: "contactus@site.com" },
    { DisplayText: "Web Master", Email: "webmaster@site.com" }
  ];
</script>
```

Here, we have defined a global variable named `CONTACT_DATA` that is an array of JSON objects. Each JSON object has a property named `DisplayText` and a property named `Email`. As we have done before, we will now need to include a definition of this variable in our `globals.d.ts` declaration file:

```
interface IContactData {
  DisplayText: string;
  Email: string;
}

declare var CONTACT_DATA: IContactData[];
```

We start with an interface definition named `IContactData` to represent the properties of an individual item in the `CONTACT_DATA` array. Each item has a `DisplayText` property that is of the type `string`, as well as an `Email` property which is also of type `string`. Our `IContactData` interface, therefore, matches the original object properties of a single item in the JSON array. We then declare a variable named `CONTACT_DATA` and set its type to be an array of the `IContactData` interfaces.

This allows us to work with the `CONTACT_DATA` variable within TypeScript. Let's now create a class to process this data, as follows:

```
class ContactLogger {
    static logContactData() {
        for (var i = 0; i < CONTACT_DATA.length; i++) {
            var contactDataItem: IContactData = CONTACT_DATA[i];
            console.log("Contact Text : "
                + contactDataItem.DisplayText
                + " Email : " + contactDataItem.Email
            );
        }
    }
}

window.onload = () => {
    ContactLogger.logContactData();
}
```

The class `ContactLogger` has a single static method named `logContactData`. Within this method, we loop through all of the items in the `CONTACT_DATA` array, using the `length` property that is inherent in all JavaScript arrays. We then create a variable named `contactDataItem` that is strongly typed to be of type `IContactData`, and assign the value of the current array item to it. Being of type `IContactData`, the `contactDataItem` will now have two properties, `DisplayText` and `Email`. We simply log these values to the console. The output of this code would be:

```
Contact Text : Help Email : help@site.com
Contact Text : Contact Us Email : contactus@site.com
Contact Text : Web Master Email : webmaster@site.com
```

Writing your own declaration file

In any development team, there will come a time when you will need to either bug-fix, or enhance a body of code that has already been written in JavaScript. If you are in this situation, then you would want to try and write new areas of code in TypeScript, and integrate them with your existing body of JavaScript. To do so, however, you will need to write your own declaration files for any existing JavaScript that you need to reuse. This may seem like a daunting and time-consuming task, but when you are faced with this situation, just remember to take small steps, and define small sections of code at a time. You will be surprised at how simple it really is.

In this section, let's assume that you need to integrate an existing helper class – one that is reused across many projects, is well-tested, and is a development team standard. This class has been implemented as a JavaScript closure as follows:

```
ErrorHelper = (function() {
  return {
    containsErrors: function (response) {
      if (!response || !response.responseText)
        return false;

      var errorValue = response.responseText;

      if (String(errorValue.failure) == "true"
          || Boolean(errorValue.failure)) {
        return true;
      }
      return false;
    },
    trace: function (msg) {
      var traceMessage = msg;
      if (msg.responseText) {
        traceMessage = msg.responseText.errorMessage;
      }
      console.log("[ " + new Date().toLocaleDateString()
        + " ] " + traceMessage);
    }
  }
})();
```

This JavaScript code snippet defines a JavaScript object named `ErrorHelper` that has two methods. The `containsErrors` method takes an object named `response` as an argument, and tests to see whether it has a property called `responseText`. If it does, it then checks to see whether the `responseText` property itself has a property named `failure`. If this `failure` property is a string containing the text "true", or if the `failure` property is a boolean with the value `true`, then this function returns `true`; in other words, we are evaluating the `response.responseText.failure` property. The `ErrorHelper` closure also has a function called `trace` that can be called with a string, or a response object similar to what the `containsErrors` function is expecting.

Unfortunately, this `ErrorHelper` function is missing a key piece of documentation. What is the structure of the object being passed into these two methods, and what properties does it have? Without some form of documentation, we are forced to reverse engineer the code to determine what the structure of the `response` object looks like. If we can find some sample usages of the `ErrorHelper` class, this may help us to guess this structure. As an example of how this `ErrorHelper` is used, consider the following JavaScript code:

```
var failureMessage = {
  responseText: {
    "failure": true,
    "errorMessage": "Unhandled Exception"
  }
};
var failureMessageString = {
  responseText: {
    "failure": "true",
    "errorMessage": "Unhandled Exception"
  }
};
var successMessage = { responseText: { "failure": false } };

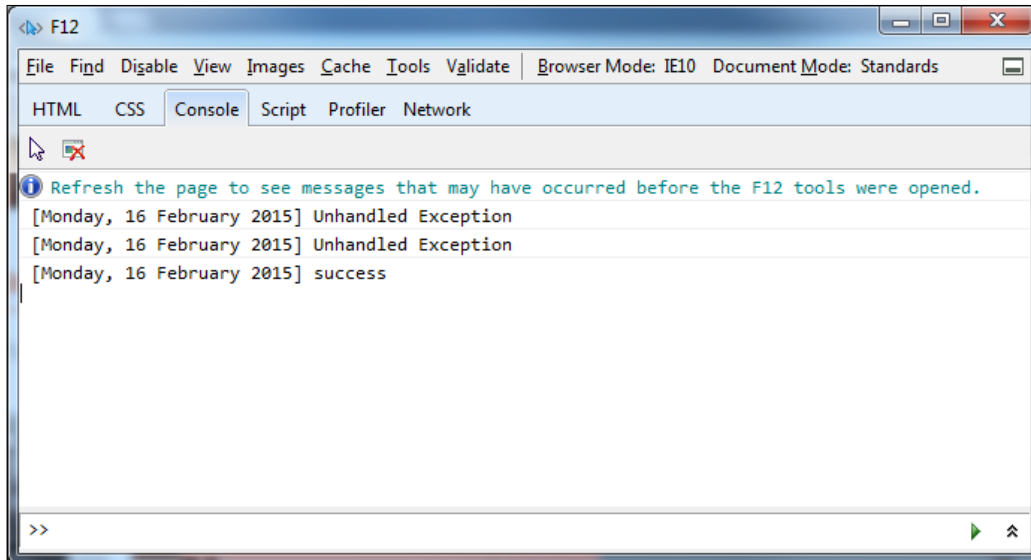
if (ErrorHelper.containsErrors(failureMessage))
  ErrorHelper.trace(failureMessage);
if (ErrorHelper.containsErrors(failureMessageString))
  ErrorHelper.trace(failureMessageString);
if (!ErrorHelper.containsErrors(successMessage))
  ErrorHelper.trace("success");
```

Here, we start with a variable named `failureMessage` that has a single property `responseText`. The `responseText` property in turn has two child properties: `failure` and `errorMessage`. Our next variable `failureMessageString` has the same structure, but defines the `responseText.failure` property to be a string, instead of a boolean value. Finally, our `successMessage` object just defines the `responseText.failure` property to be `false`, but it does not have an `errorMessage` property.



In JavaScript JSON format, property names are required to have quotes around them, whereas in JavaScript these optional. Therefore, the structure `{"failure" : true}` is syntactically equivalent to the structure `{failure : true}`.

The last couple of lines of the preceding code snippet show how the `ErrorHelper` closure is used. All we need to do is call the `ErrorHelper.containsErrors` method with our variable, and if the result is `true`, log the message to the console via the `ErrorHelper.trace` function. Our output would be as follows:



ErrorHelper console output

The module keyword

To test this JavaScript `ErrorHelper` closure using TypeScript, we will need an HTML page that includes both the `ErrorHelper.js` file, and a TypeScript generated JavaScript file. Assuming that our TypeScript file is called `ErrorHelperTypeScript.ts`, our HTML page would then be as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>specify.
  <title></title>
  <script src="ErrorHelper.js"></script>
  <script src="ErrorHelperTypeScript.js"></script>
</head>
<body>

</body>
</html>
```

This HTML is very simple, and includes both the existing `ErrorHelper.js` JavaScript file, as well as the TypeScript generated `ErrorHelperTypeScript.js` file.

Within the `ErrorHelperTypeScript.ts` file, let's use the `ErrorHelper` as follows:


```
window.onload = () => {
    var failureMessage = {
        responseText: { "failure": true,
            "errorMessage": "Unhandled Exception" }
    };

    if (ErrorHelper.containsErrors(failureMessage))
        ErrorHelper.trace(failureMessage);
}
```

This code snippet shows a stripped down version of our original JavaScript sample. In fact, we can just copy and paste the original JavaScript code into our TypeScript file. We first create a `failureMessage` object with the correct properties, and then simply call the `ErrorHelper.containsErrors` method, and the `ErrorHelper.trace` method. If we were to compile our TypeScript file at this stage, we would receive the following error:

```
error TS2095: Build: Could not find symbol 'ErrorHelper'.
```

This error is indicating that there is no valid TypeScript type named `ErrorHelper`, even though we have the full source of `ErrorHelper` in our JavaScript file. TypeScript by default, will look through all the TypeScript files in our project to find class definitions, but it will not parse JavaScript files. We will need a new TypeScript definition file in order to correctly compile this code.

 This definition file is not included in the HTML file at all; it is only used by the TypeScript compiler and does not generate any JavaScript.

Without a set of helpful documentation on our `ErrorHelper` class, we will need to reverse-engineer a TypeScript definition purely by reading the source code. This is obviously not an ideal situation, and is not recommended, but at this stage, it is all we can do. In these situations, the best starting point is simply to look at the usage samples and work our way up from there.

Looking at the usage of the `ErrorHelper` closure in JavaScript, there are two key pieces that should be included in our declaration file. The first is a set of function definitions for the `containsErrors` and `trace` functions. The second is a set of interfaces to describe the structure of the response object that the `ErrorHelper` closure relies upon. Let's start with the function definitions, and create a new TypeScript file named `ErrorHelper.d.ts` with the following code:

```
declare module ErrorHelper {  
    function containsErrors(response);  
    function trace(message);  
}
```

This declaration file starts with the `declare` keyword that we have seen before, but then uses a new TypeScript keyword: `module`. The `module` keyword must be followed by a module name, which in this case is `ErrorHelper`. This module name must match the closure name from the original JavaScript that we are describing. In all of our usages of the `ErrorHelper`, we have always pre-fixed the functions `containsErrors` and `trace` with the closure name `ErrorHelper` itself. This module name is also known as a namespace. If we had another class named `AjaxHelper` that also included a `containsErrors` function, we would be able to distinguish between the `AjaxHelper.containsErrors` and the `ErrorHelper.containsErrors` functions by using these namespaces, or module names.

The second line of the preceding code snippet indicates that we are defining a function called `containsErrors` that takes one parameter. The third line of this module declaration indicates that we are defining another function named `trace` that takes one parameter. With this definition in place, our TypeScript code sample will compile correctly.

Interfaces

Although we have correctly defined the two functions that are available to users of the `ErrorHelper` closure, we are missing the second piece of information about the functions available on the `ErrorHelper` closure—the structure of the response argument. We are not strongly typing the arguments for either of the `containsErrors` or `trace` functions. At this stage, our TypeScript code can pass anything into these two functions because it does not have a definition for the response or message arguments. We know, however, that both these functions query these arguments for a specific structure. If we pass in an object that does not conform to this structure, then our JavaScript code will cause runtime errors.

To solve this problem and to make our code more stable, let's define an interface for these parameters:

```
interface IResponse {
    responseText: IFailureMessage;
}

interface IFailureMessage {
    failure: boolean;
    errorMessage: string;
}
```

We start with an interface named `IResponse` that has a single property of `responseText` — the same name as the original JSON object. This `responseText` property is strongly typed to be of type `IFailureMessage`. The `IFailureMessage` interface is strongly typed to have two properties: `failure`, which is a `boolean`, and `errorMessage`, which is of type `string`. These interfaces correctly describe the proper structure of the `response` argument for the `containsErrors` function. We can now modify our original declaration for the `containsErrors` function to use this interface on the `response` argument as follows:

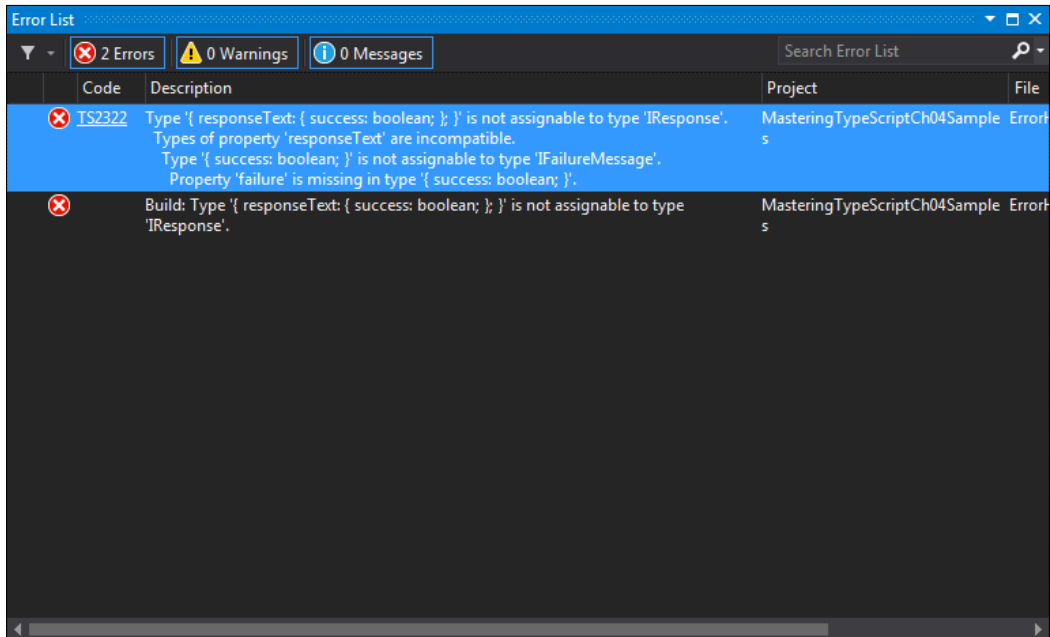
```
declare module ErrorHandler {
    function containsErrors(response: IResponse);
    function trace(message);
}
```

The function definition for `containsErrors` now strongly types the `response` argument to be of type `IResponse`, which we defined earlier. This modification to the definition file will now force any further usage of the `containsErrors` function to send in a valid argument that conforms to the `IResponse` structure. Let's write some intentionally incorrect TypeScript code and see what happens:

```
var anotherFailure : IResponse = { responseText: { success: true } };

if (ErrorHandler.containsErrors(anotherFailure))
    ErrorHandler.trace(anotherFailure);
```

We start by creating a variable named `anotherFailure` and specify its type to be of type `IResponse`. Even though we are using a definition file to define this interface, the rules that are applied by the TypeScript compiler, are no different to what we have seen before. The first line in this code snippet will generate the following error:



Compile errors for an incorrect `responseText` object

As can be seen from this fairly verbose but informative error message, the structure of the `anotherFailure` variable is causing all the errors. Even though we have correctly referenced the `responseText` property of `IResponse`, the `responseText` property is strongly typed to be of type `IFailureMessage`, which requires both a `failure` property and an `errorMessage` property; hence the error.

We can fix these errors by including the required properties of `failure` and `errorMessage` within the variable `anotherFailure`:

```
var anotherFailure: IResponse = {
  responseType: {
    failure: false, errorMessage: "", success: true
  }
};
```

Our TypeScript now compiles correctly. The variable `anotherFailure` now has all of the required properties in order to use the `ErrorHelper` functions correctly. By creating a strongly typed declaration file for the existing `ErrorHelper` class, we can ensure that any further TypeScript usage of the existing `ErrorHelper` JavaScript closure will not generate runtime errors.

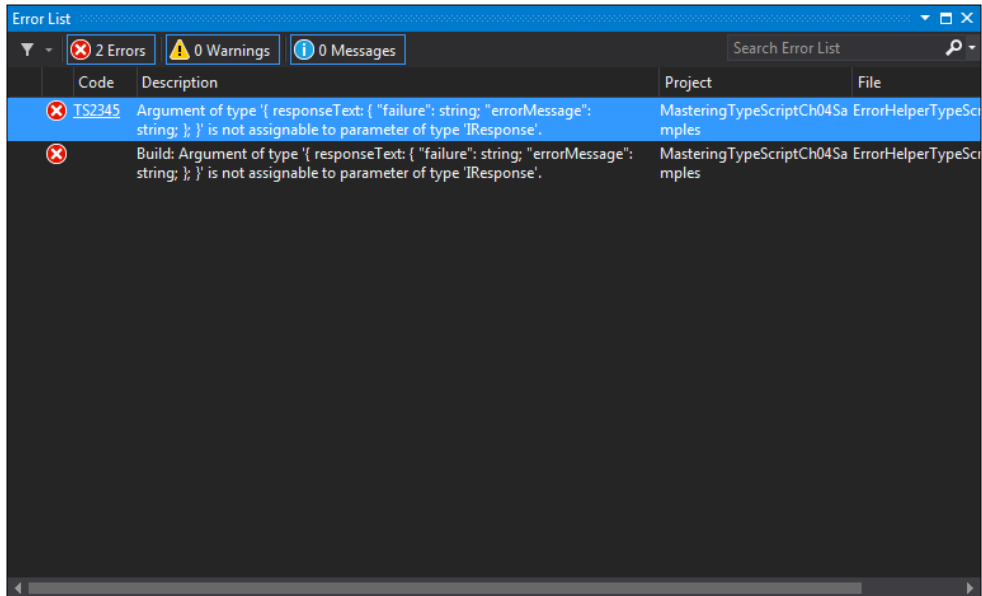
Function overrides

We are not quite finished with the declaration file for the `ErrorHelper` just yet. If we take a look at the original JavaScript usage of the `ErrorHelper`, we will notice that the `containsErrors` function also allows for the `failure` property of `responseText` to be a string:

```
var failureMessageString = {
  responseText: {
    "failure": "true",
    "errorMessage": "Error Message" }
};

if (ErrorHelper.containsErrors(failureMessageString))
  ErrorHelper.trace(failureMessage);
```

If we compile this code now, we will get the following compile error:



Compile errors for multiple definitions of `responseText`

In the preceding definition of the variable `failureMessageString`, the type of the "failure" property is "true", which is of type `string`, and not `true`, which is of type `boolean`. In order to allow for this variant on the original `IFailureMessage` interface, we will need to modify our declaration file. Firstly, we will need two new interfaces that specify the `failure` property to be of type `string`:

```
interface IResponseString {
  responseText: IFailureMessageString;
}

interface IFailureMessageString {
  failure: string;
  errorMessage: string;
}
```

The `IResponseString` interface is almost identical to the `IResponse` interface, except that it uses the `IFailureMessageString` type for the property `responseText`. This `IFailureMessageString` interface is also almost identical to the original `IFailureMessage` interface, except that the `failure` property is of type `string`. We will now need to modify our declaration file to allow both call signatures on the `containsErrors` function:

```
declare module ErrorHandler {
  function containsErrors(response: IResponse);
  function containsErrors(response: IResponseString);
  function trace(message);
}
```

As with interface and class definitions, modules also allow for function overrides. The module `ErrorHandler` now has one function definition for `containsErrors` that uses the original `IResponse` interface, and a second function definition that uses the new `IResponseString` interface. This new version of the module definition will allow both variants of the `failure` message structure to compile correctly.

We can also take advantage of union types in this example, and simplify our previous declaration for the `containsErrors` function to a single definition:

```
declare module ErrorHandler {
  function containsErrors(response: IResponse | IResponseString);
  function trace(message: string);
}
```


Rounding out our definition file

We can now focus our attention on the `trace` function. The `trace` function can accept both versions of the `IResponse` interface, or it can simply accept a string. Let's update the definition file for the `trace` function signatures:

```
declare module ErrorHandler {
    function containsErrors(response: IResponse | IResponseString);
    function trace(message: string | IResponse | IResponseString);
}
```

Here, we have updated the `trace` function to allow three different variants of the message type—a normal `string`, an `IResponse` type, or an `IResponseString` type.

This completes our definition file for the `ErrorHandler` JavaScript class.

Module merging

As we now know, the TypeScript compiler will automatically search through all the `.d.ts` files in your project to pick up declaration files. If these declaration files contain the same module name, the TypeScript compiler will merge these two declaration files and use a combined version of the module declarations.

If we have a file named `MergedModule1.d.ts` that contains the following definition:

```
declare module MergedModule {
    function functionA();
}
```

And a second file named `MergedModule2.d.ts` that contains the following definition:

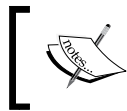
```
declare module MergedModule {
    function functionB();
}
```

The TypeScript compiler will merge these two modules as if they were a single definition:

```
declare module MergedModule {
    function functionA();
    function functionB();
}
```

This will allow both `functionA` and `functionB` to be valid functions of the same `MergedModule` namespace and allow the following usage:

```
MergedModule.functionA();
MergedModule.functionB();
```



Modules can also merge with interfaces, classes, and enums. Classes, however, cannot merge with other classes, variables, or interfaces.

Declaration Syntax Reference

When creating declaration files and using the `module` keyword, there are a number of rules that can be used to mix and match definitions. We have covered one of them already – function overrides. As a TypeScript programmer, you will generally only write module definitions every now and then, and on occasion, need to add a new definition to an existing declaration file.

This section, therefore, is designed to be a quick reference guide to this declaration file syntax, or a cheat-sheet. Each section contains a description of the module definition rule, a JavaScript syntax snippet, and then the equivalent TypeScript declaration file syntax.

To use this reference section, simply match the JavaScript that you are trying to emulate from the JavaScript syntax section, and then write your declaration file with the equivalent definition syntax. We will start with the function overrides syntax as an example:

Function overrides

Declaration files can include multiple definitions for the same function. If the same JavaScript function can be called with different types, you will need to declare a function override for each variant of the function.

The JavaScript syntax

```
trace("trace a string");
trace(true);
trace(1);
trace({ id: 1, name: "test" });
```

The declaration file syntax

```
declare function trace(arg: string | number | boolean );  
declare function trace(arg: { id: number; name: string });
```



Each function definition must have a unique function signature.

Nested namespaces

Module definitions can contain nested module definitions, which then translate to nested namespaces. If your JavaScript uses namespaces, then you will need to define nested module declarations to match the JavaScript namespaces.

The JavaScript syntax

```
FirstNamespace.SecondNamespace.ThirdNamespace.log("test");
```

The declaration file syntax

```
declare module FirstNamespace {  
  module SecondNamespace {  
    module ThirdNamespace {  
      function log(msg: string);  
    }  
  }  
}
```

Classes

Class definitions are allowed within module definitions. If your JavaScript uses classes, or the new operator, then the new-able classes will need to be defined in your declaration file.

The JavaScript syntax

```
var myClass = new MyClass();
```

The declaration file syntax

```
declare class MyClass {  
}
```

Class namespaces

Class definitions are allowed within nested module definitions. If your JavaScript classes have a preceding namespace, you will need to declare nested modules to match the namespaces first, and then you can declare classes within the correct namespace.

The JavaScript syntax

```
var myNestedClass = new OuterName.InnerName.NestedClass();
```

The declaration file syntax

```
declare module OuterName {
  module InnerName {
    class NestedClass {}
  }
}
```

Class constructor overloads

Class definitions can contain constructor overloads. If your JavaScript classes can be constructed using different types, or with multiple parameters, you will need to list each of these variants in your declaration file as constructor overloads.

The JavaScript syntax

```
var myClass = new MyClass();
var myClass2 = new MyClass(1, "test");
```

The declaration file syntax

```
declare class MyClass {
  constructor(id: number, name: string);
  constructor();
}
```

Class properties

Classes can contain properties. You will need to list each property of your class within your class declaration.

The JavaScript syntax

```
var classWithProperty = new ClassWithProperty();
classWithProperty.id = 1;
```

The declaration file syntax

```
declare class ClassWithProperty {
  id: number;
}
```

Class functions

Classes can contain functions. You will need to list each function of your JavaScript class within your class declaration, in order for the TypeScript compiler to accept calls to these functions.

The JavaScript syntax

```
var classWithFunction = new ClassWithFunction();
classWithFunction.functionToRun();
```

The declaration file syntax

```
declare class ClassWithFunction {
  functionToRun(): void;
}
```



Functions or properties that are considered as private do not need to be exposed via the declaration file, and can simply be omitted.

Static properties and functions

Class methods and properties can be static. If your JavaScript functions or properties can be called without needing an instance of an object to work with, then these properties or functions will need to be marked as static.

The JavaScript syntax

```
StaticClass.staticId = 1;
StaticClass.staticFunction();
```

The declaration file syntax

```
declare class StaticClass {
  static staticId: number;
  static staticFunction();
}
```

Global functions

Functions that do not have a namespace prefix can be declared in the global namespace. If your JavaScript defines global functions, these will need to be declared without a namespace.

The JavaScript syntax

```
globalLogError("test");
```

The declaration file syntax

```
declare function globalLogError(msg: string);
```

Function signatures

A function can use a function signature as a parameter. JavaScript functions that use callback functions or anonymous functions, will need to be declared with the correct function signature.

The JavaScript syntax

```
describe("test", function () {
  console.log("inside the test function");
});
```

The declaration file syntax

```
declare function describe(name: string, functionDef: () => void);
```

Optional properties

Classes or functions can contain optional properties. Where JavaScript object parameters are not mandatory, these will need to be marked as optional properties in the declaration.

The JavaScript syntax

```
var classWithOpt = new ClassWithOptionals();
var classWithOpt1 = new ClassWithOptionals({ id: 1 });
var classWithOpt2 = new ClassWithOptionals({ name: "first" });
var classWithOpt3 = new ClassWithOptionals({ id: 2, name: "second"
});
```

The declaration file syntax

```
interface IOptionalProperties {
    id?: number;
    name?: string;
}

declare class ClassWithOptionals {
    constructor(options?: IOptionalProperties);
}
```

Merging functions and modules

A function definition with a specific name can be merged with a module definition of the same name. This means that if your JavaScript function can be called with parameters and also has properties, then you will need to merge a function with a module.

The JavaScript syntax

```
fnWithProperty(1);
fnWithProperty.name = "name";
```

The declaration file syntax

```
declare function fnWithProperty(id: number);
declare module fnWithProperty {
    var name: string;
}
```

Summary

In this chapter, we have outlined what you need to know in order to write and use your own declaration files. We discussed JavaScript global variables in rendered HTML and how to access them in TypeScript. We then moved on to a small JavaScript helper function and wrote our own declaration file for this JavaScript. We finished off the chapter by listing a few module definition rules, highlighting the required JavaScript syntax, and showing what the equivalent TypeScript declaration syntax would be. In the next chapter, we will look at how to use existing third-party JavaScript libraries, and how to import existing declaration files for these libraries into your TypeScript projects.

5

Third Party Libraries

Our TypeScript development environment would not amount to much if we were not able to re-use the myriad of existing JavaScript libraries, frameworks and general goodness. As we have seen, however, in order to use a particular third party library with TypeScript, we will first need a matching definition file.

Soon after TypeScript was released, Boris Yankov set up a github repository to house TypeScript definition files for third party JavaScript libraries. This repository, named DefinitelyTyped (<https://github.com/borisyankov/DefinitelyTyped>) quickly became very popular, and is currently the place to go for high-quality definition files. DefinitelyTyped currently has over 700 definition files, built up over time from hundreds of contributors from all over the world. If we were to measure the success of TypeScript within the JavaScript community, then the DefinitelyTyped repository would be a good indication of how well TypeScript has been adopted. Before you go ahead and try to write your own definition files, check the DefinitelyTyped repository to see if there is one already available.

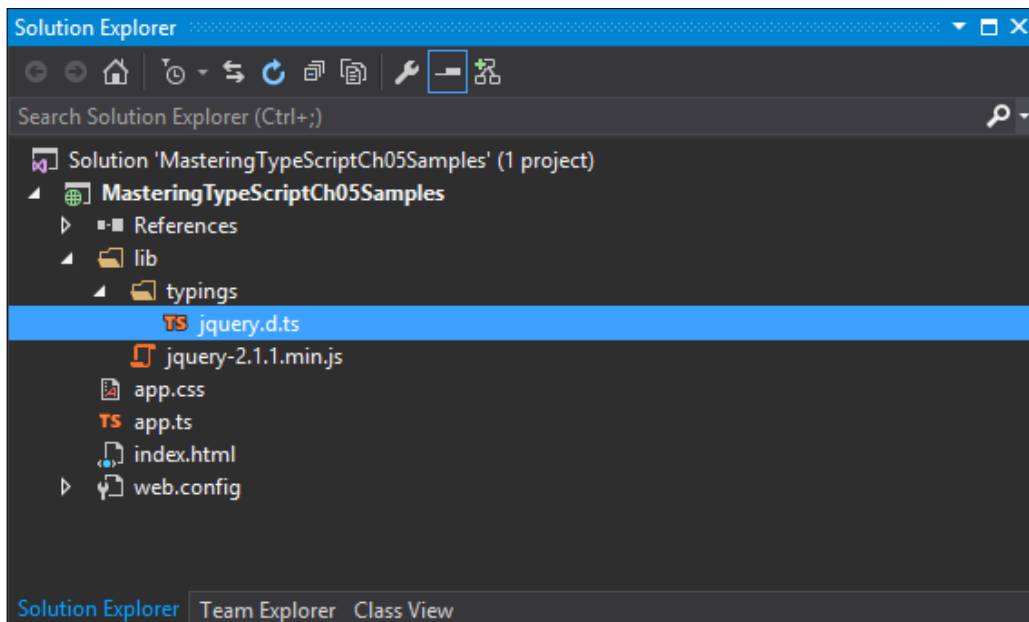
In this chapter, we will have a closer look at using these definition files, and cover the following topics:

- Downloading definition files
- Using NuGet within Visual Studio
- Using TypeScript Definition manager (TSD)
- Choosing a JavaScript Framework
- Using TypeScript with Backbone
- Using TypeScript with Angular
- Using TypeScript with ExtJs

Downloading definition files

The simplest method of including a definition file within your TypeScript project is to download the matching `.d.ts` file from DefinitelyTyped. This is a simple matter of finding the relevant file, and downloading the raw content. Let's assume that we wanted to start using jQuery within our project. We have found and downloaded the jQuery JavaScript library (v2.1.1), and included the relevant files within our project, under a directory named `lib`. To download the declaration file, simply browse to the `jquery` directory on DefinitelyTyped (<https://github.com/borisyankov/DefinitelyTyped/tree/master/jquery>), and then click on the `jquery.d.ts` file. This will open up a GitHub page with an editor view of the file. On the menu bar of this editor view, click on the **Raw** button. This will download the `jquery.d.ts` file, and allow you to save it within your project directory structure. Create a new directory under the `lib` folder called `typings`, and save the `jquery.d.ts` file there.

Your project file should look something like this:



Visual Studio project structure with a downloaded `jquery.d.ts` file

We can now modify our `index.html` file to include the `jquery` JavaScript file, and begin writing TypeScript code that targets the `jQuery` library. Our `index.html` file will need to be modified as follows:

```
<!DOCTYPE html>

<html lang="en">
<head>
  <meta charset="utf-8" />
  <title>TypeScript HTML App</title>
  <link rel="stylesheet" href="app.css" type="text/css" />
  <script src="/lib/jquery-2.1.1.min.js"></script>
  <script src="app.js"></script>
</head>
<body>
  <h1>TypeScript HTML App</h1>

  <div id="content"></div>
</body>
</html>
```

The first `<script>` tag of this `index.html` file now includes a link to `jquery-2.1.1.min.js`, and the second `<script>` tag includes a link to the TypeScript generated `app.js`. Open up the `app.ts` TypeScript file, delete the existing source, and replace it with the following `jQuery` code:

```
$(document).ready(() => {
  $("#content").html("<h1>Hello World !</h1>");
});
```

This snippet starts by defining an anonymous function to execute on the `jQuery` event of `document.ready`. The `document.ready` function is similar to the `window.onload` function we have been using previously, and will execute once `jQuery` has initialized. The second line of this snippet simply gets a handle to the DOM element named `content` using `jQuery` selector syntax, and then calls the `html` function to set its HTML value.

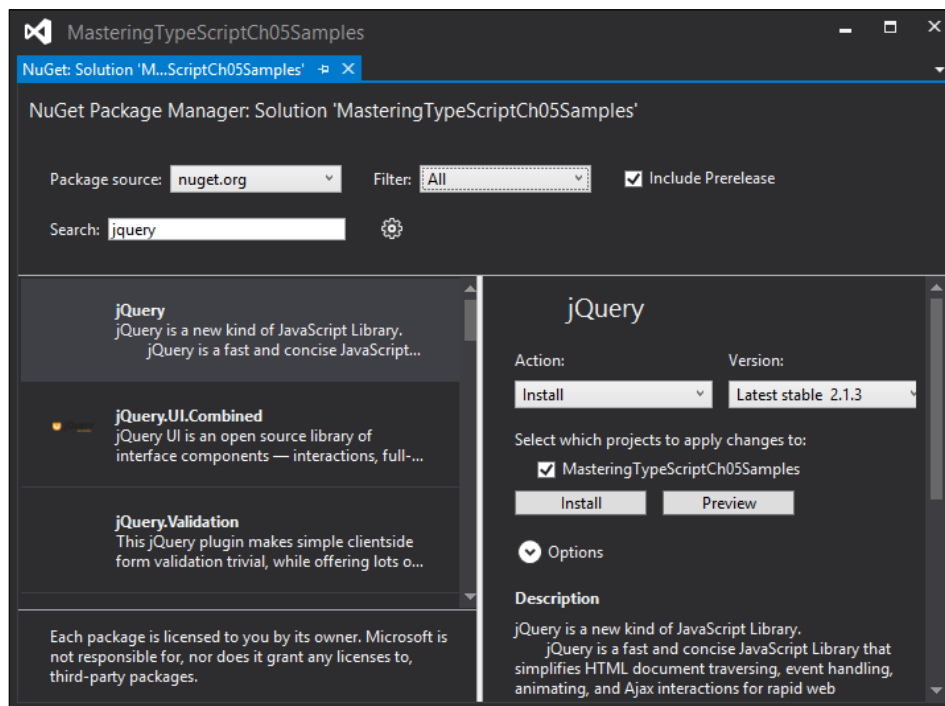
The `jquery.d.ts` file that we downloaded is providing us with the relevant module declarations that we need in order to compile `jQuery` within TypeScript.

Using NuGet

NuGet is a popular package management platform that will download required external libraries, and automatically include them in within your Visual Studio or WebMatrix project. It can be used for external libraries that are packaged as DLLs – such as StructureMap – or it can be used for JavaScript libraries and declaration files. NuGet is also available as a command-line utility.

Using the Extension Manager

To use the NuGet package manager dialog within Visual Studio, select the **Tools** option on the main toolbar, then select **NuGet Package Manager**, and finally select **Manage NuGet Packages for Solution**. This brings up the NuGet package manager dialog. On the left-hand side of the dialog, click on **Online**. The NuGet dialog will then query the NuGet website and show a list of available packages. At the top right of the screen is a **search** box. Click within the **search** box, and type `jquery` to show all packages available within NuGet for jQuery, as shown in the following screenshot:



NuGet Package manager dialog with results from a query on jQuery

Each package will have an **Install** button highlighted when you select the package in the **search results** panel. When a package is selected, the right-hand pane will show more details about the NuGet package in question. Note that the project details panel also shows the Version of the package that you are about to install. Clicking on the **Install** button will download relevant files – as well as any dependencies – and include them automatically within your project.



The installation directory that NuGet uses for JavaScript files is in fact called `Scripts` – and not the `lib` directory that we created earlier. NuGet uses the `Scripts` directory as a standard, so any packages that contain JavaScript will install the relevant JavaScript files into the `Scripts` directory.

Installing declaration files

You will find that most declaration files that are found on the DefinitelyTyped GitHub repository have a corresponding NuGet package. These packages are named `<library>.TypeScript.DefinitelyTyped`, as a standard naming convention. If we type `jquery typescript` into the search box, we will see a list of these DefinitelyTyped packages returned. The NuGet package we are looking for is named **jquery.TypeScript.DefinitelyTyped**, created by **Jason Jarret**, and is, at the time of writing, at version 1.4.0.



The DefinitelyTyped packages have their own internal version number, and these version numbers do not necessarily match the version of the JavaScript library that you are using. For example, the jQuery package is at version 2.1.1, but the corresponding TypeScript definition package shows a version number of 1.4.0.

Installing the `jquery.TypeScript.DefinitelyTyped` package will create a `typings` directory under the `Scripts` directory, and then include the `jquery.d.ts` definition file. This directory naming standard has been adopted by the various NuGet package authors.

Using the Package Manager Console

Visual Studio also has a command-line version of the NuGet package manager available as a console application, and is also integrated into Visual Studio. Clicking on **Tools**, then **NuGet Package Manager**, and finally on **Package Manager Console**, will bring up a new Visual Studio window, and initialize the NuGet command line interface. The command line version of NuGet has a number of features that are not included in the GUI version. Type `get -help NuGet` to see the list of top-level command line arguments that are available.

Installing packages

To install a NuGet package from the console command line, simply type `install-package <packageName>`. As an example, to install the `jquery.TypeScript.DefinitelyTyped` package, simply type:

```
Install-Package jquery.TypeScript.DefinitelyTyped
```

This command will connect to the NuGet server, and download and install the package into your project.



On the toolbar of the **Package Manager Console** window are two dropdown lists, **Package Source** and **Default Project**. If your Visual Studio solution has multiple projects, you will need to select the correct project for NuGet to install the package into from the **Default Project** dropdown.

Searching for package names

Searching for package names from the command line is accomplished with the `Get-Package -ListAvailable` command. This command takes a `-Filter` parameter which acts as the search criteria. As an example, to find available packages that include the `definitelytyped` search string, run the following command:

```
Get-Package -ListAvailable -Filter definitelytyped
```

Installing a specific version

There are some JavaScript libraries that are not compatible with jQuery version 2.x, and will require a version of jQuery that is in the 1.x range. To install a specific version of a NuGet package, we will need to specify the `-Version` parameter from the command line. To install the `jquery v1.11.1` package, as an example, run the following from the command line:

```
Install-Package jQuery -Version 1.11.1
```



NuGet will either upgrade or downgrade the version of the package you are installing, if it finds another version already installed within your project. In the preceding example, we had already installed the latest version of jQuery (2.1.1) within our project, so NuGet will first remove jQuery 2.1.1 before installing jQuery 1.11.1.

Using TypeScript Definition Manager

If you are using Node as your TypeScript development environment, then you may consider using the **TypeScript Definition Manager** for DefinitelyTyped (TSD at <http://definitelytyped.org/tsd/>). TSD offers similar functionality to the NuGet Package Manager, but is specifically geared towards TypeScript definitions that are part of the DefinitelyTyped GitHub repository.

To install TSD, use npm as follows:

```
npm install tsd@next -g
```

This will install tsd prerelease v0.6.x.



At the time of writing, you will need v0.6.x and up in order to use the install keyword from the command line. If you simply type `npm install tsd -g`, then npm will install v0.5.x, which does not include the install keyword.

Querying for packages

TSD allows for querying the package repository using the query keyword. To search for the jquery definition files, type the following:

```
tsd query jquery
```

The preceding command will search the DefinitelyTyped repository for any definition files named `jquery.d.ts`. Since there is only one, the results returned from the search would be:

```
Jquery / jquery
```


Using wildcards

TSD also allows for the use of the asterisk `*` as a wildcard. To search for DefinitelyTyped declaration files that start with `jquery`, type the following:

```
tsd query jquery.*
```

This `tsd` command will search through the repository, and return results for declaration files that start with `jQuery`.

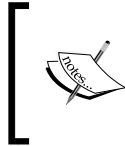
Installing definition files

To install a definition file, use the `install` keyword as follows:

```
tsd install jquery
```

This command will download the `jquery.d.ts` file into the following directory:

```
\typings\jquery\jquery.d.ts
```



TSD will create the `\typings` directory based on the current directory where `tsd` was run, so make sure that you navigate to the same base directory in your project whenever you use TSD from the command line.

Using third party libraries

In this section of the chapter, we will begin to explore some of the more popular third party JavaScript libraries, their declaration files, and how to write compatible TypeScript for each of these frameworks. We will compare Backbone, Angular, and ExtJs, which are all frameworks for building rich client-side JavaScript applications. During our discussion, we will see that some frameworks are highly compliant with the TypeScript language and its features, some are partially compliant, and some have very low compliance.

Choosing a JavaScript framework

Choosing a JavaScript framework or library to develop Single Page Applications is a difficult and sometimes daunting task. It seems that there is a new framework appearing every other month, promising more and more functionality for less and less code.

To help developers compare these frameworks, and make an informed choice, Addy Osmani wrote an excellent article, named *Journey Through the JavaScript MVC Jungle*. (<http://www.smashingmagazine.com/2012/07/27/journey-through-the-javascript-mvc-jungle/>).

In essence, his advice is simple – it's a personal choice – so try some frameworks out, and see what best fits your needs, your programming mindset, and your existing skill set. The **TodoMVC** project (<http://todomvc.com>), which Addy started, does an excellent job of implementing the same application in a number of MV* JavaScript frameworks. This really is a reference site for digging into a fully working application, and comparing for yourself the coding techniques and styles of different frameworks.

Again, depending on the JavaScript library that you are using within TypeScript, you may need to write your TypeScript code in a specific way. Bear this in mind when choosing a framework – if it is difficult to use with TypeScript, then you may be better off looking at another framework with better integration. If it is easy and natural to work with the framework in TypeScript, then your productivity and overall development experience will be much better.

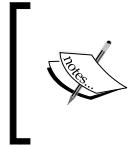
In this section, we will look at some of the popular JavaScript libraries, along with their declaration files, and see how to write compatible TypeScript. The key thing to remember is that TypeScript generates JavaScript – so if you are battling to use a third party library, then crack open the generated JavaScript and see what the JavaScript code looks like that TypeScript is emitting. If the generated JavaScript matches the JavaScript code samples in the library's documentation, then you are on the right track. If not, then you may need to modify your TypeScript until the compiled JavaScript starts matching up with the samples.

When trying to write TypeScript code for a third party JavaScript framework – particularly if you are working off the JavaScript documentation – your initial foray may just be one of trial and error. Along the way, you may find that you need to write your TypeScript in a specific way in order to match this particular third party library. The rest of this chapter shows how three different libraries require different ways of writing TypeScript.

Backbone

Backbone is a popular JavaScript library that gives structure to web applications by providing models, collections and views, amongst other things. Backbone has been around since 2010, and has gained a very large following, with a wealth of commercial websites using the framework. According to [Infoworld.com](http://infoworld.com), Backbone has over 1,600 Backbone related projects on GitHub that rate over 3 stars – meaning that it has a vast ecosystem of extensions and related libraries.

Let's take a quick look at Backbone written in TypeScript.



To follow along with the code in your own project, you will need to install the following NuGet packages: `backbone.js` (currently at v1.1.2), and `backbone.TypeScript.DefinitelyTyped` (currently at version 1.2.3).

Using inheritance with Backbone

From the Backbone documentation, we find an example of creating a `Backbone.Model` in JavaScript as follows:

```
var Note = Backbone.Model.extend(  
  {  
    initialize: function() {  
      alert("Note Model JavaScript initialize");  
    },  
    author: function () { },  
    coordinates: function () { },  
    allowedToEdit: function(account) {  
      return true;  
    }  
  }  
);
```

This code shows a typical usage of Backbone in JavaScript. We start by creating a variable named `Note` that extends (or derives from) `Backbone.Model`. This can be seen with the `Backbone.Model.extend` syntax. The Backbone `extend` function uses JavaScript object notation to define an object within the outer curly braces `{ ... }`. In the preceding code, this object has four functions: `initialize`, `author`, `coordinates` and `allowedToEdit`.

According to the Backbone documentation, the `initialize` function will be called once a new instance of this class is created. In our preceding sample, the `initialize` function simply creates an **alert** to indicate that the function was called. The `author` and `coordinates` functions are blank at this stage, with only the `allowedToEdit` function actually doing something: `return true`.

If we were to simply copy and paste the above JavaScript into a TypeScript file, we would generate the following compile error:

```
Build: 'Backbone.Model.extend' is inaccessible.
```

When working with a third party library, and a definition file from DefinitelyTyped, our first port of call should be to see if the definition file may be in error. After all, the JavaScript documentation says that we should be able to use the `extend` method as shown, so why is this definition file causing an error? If we open up the `backbone.d.ts` file, and then search to find the definition of the class `Model`, we will find the cause of the compilation error:

```
class Model extends ModelBase {

    /**
     * Do not use, prefer TypeScript's extend functionality.
     */
    private static extend(

        properties: any, classProperties?: any): any;
```

This declaration file snippet shows some of the definition of the Backbone `Model` class. Here, we can see that the `extend` function is defined as `private static`, and as such, it will not be available outside the `Model` class itself. This, however, seems contradictory to the JavaScript sample that we saw in the documentation. In the preceding comment on the `extend` function definition, we find the key to using Backbone in TypeScript: prefer TypeScript's `extend` functionality.

This comment indicates that the declaration file for Backbone is built around TypeScript's `extends` keyword – thereby allowing us to use natural TypeScript inheritance syntax to create Backbone objects. The TypeScript equivalent to this code, therefore, must use the `extends` TypeScript keyword to derive a class from the base class `Backbone.Model`, as follows:

```
class Note extends Backbone.Model {
    initialize() {
        alert("Note model Typescript initialize");
    }
    author() { }
    coordinates() { }
    allowedToEdit(account) {
        return true;
    }
}
```

We are now creating a class definition named `Note` that extends the `Backbone.Model` base class. This class then has the functions `initialize`, `author`, `coordinates` and `allowedToEdit`, similar to the previous JavaScript version. Our Backbone sample will now compile and run correctly.

With either of these versions, we can create an instance of the `Note` object by including the following script within an HTML page:

```
<script type="text/javascript">
  $(document).ready( function () {
    var note = new Note();
  });
</script>
```

This JavaScript sample simply waits for the jQuery `document.ready` event to be fired, and then creates an instance of the `Note` class. As documented earlier, the `initialize` function will be called when an instance of the class is constructed, so we would see an alert box appear when we run this in a browser.

All of Backbone's core objects are designed with inheritance in mind. This means that creating new Backbone collections, views and routers will use the same `extends` syntax in TypeScript. Backbone, therefore, is a very good fit for TypeScript, because we can use natural TypeScript syntax for inheritance to create new Backbone objects.

Using interfaces

As Backbone allows us to use TypeScript inheritance to create objects, we can just as easily use TypeScript interfaces with any of our Backbone objects as well. Extracting an interface for the `Note` class above would be as follows:

```
interface INoteInterface {
  initialize();
  author();
  coordinates();
  allowedToEdit(account: string);
}
```

We can now update our `Note` class definition to implement this interface as follows:

```
class Note extends Backbone.Model implements INoteInterface {
  // existing code
}
```

Our class definition now implements the `INoteInterface` TypeScript interface. This simple change protects our code from being modified inadvertently, and also opens up the ability to work with core Backbone objects in standard object-oriented design patterns. We could, if we needed to, apply the Factory Pattern described in *Chapter 3, Interfaces, Classes and Generics*, to return a particular type of Backbone Model – or any other Backbone object for that matter.

Using generic syntax

The declaration file for Backbone has also added generic syntax to some class definitions. This brings with it further strong typing benefits when writing TypeScript code for Backbone. Backbone collections (surprise, surprise) house a collection of Backbone models, allowing us to define collections in TypeScript as follows:

```
class NoteCollection extends Backbone.Collection<Note> {
  model = Note;
  //model: Note; // generates compile error
  //model: { new (): Note }; // ok
}
```

Here, we have a `NoteCollection` that derives from, or extends a `Backbone.Collection`, but also uses generic syntax to constrain the collection to handle only objects of type `Note`. This means that any of the standard collection functions such as `at()` or `pluck()` will be strongly typed to return `Note` models, further enhancing our type safety and Intellisense.

Note the syntax used to assign a type to the internal `model` property of the collection class on the second line. We cannot use the standard TypeScript syntax `model: Note`, as this causes a compile time error. We need to assign the `model` property to a the class definition, as seen with the `model=Note` syntax, or we can use the `{ new(): Note }` syntax as seen on the last line.

Using ECMAScript 5

Backbone also allows us to use ECMAScript 5 capabilities to define getters and setters for `Backbone.Model` classes, as follows:

```
interface ISimpleModel {
  Name: string;
  Id: number;
}
class SimpleModel extends Backbone.Model implements ISimpleModel {
  get Name() {
    return this.get('Name');
  }
  set Name(value: string) {
    this.set('Name', value);
  }
}
```

```
    get Id() {
        return this.get('Id');
    }
    set Id(value: number) {
        this.set('Id', value);
    }
}
```

In this snippet, we have defined an interface with two properties, named `ISimpleModel`. We then define a `SimpleModel` class that derives from `Backbone.Model`, and also implements the `ISimpleModel` interface. We then have ES 5 getters and setters for our `Name` and `Id` properties. Backbone uses class attributes to store model values, so our getters and setters simply call the underlying `get` and `set` methods of `Backbone.Model`.

Backbone TypeScript compatibility

As we have seen, Backbone allows us to use all of TypeScript's language features within our code. We can use classes, interfaces, inheritance, generics and even ECMAScript 5 properties. All of our classes also derive from base Backbone objects. This makes Backbone a highly compatible library for building web applications with TypeScript. We will explore more of the Backbone framework in later chapters.

Angular

AngularJs (or just Angular) is also a very popular JavaScript framework, and is maintained by Google. Angular takes a completely different approach to building JavaScript SPA's, introducing an HTML syntax that the running Angular application understands. This provides the application with two-way data binding capabilities, which automatically synchronizes models, views and the HTML page. Angular also provides a mechanism for **Dependency Injection (DI)**, and uses services to provide data to your views and models.

Let's take a look at a sample from the Angular Tutorial, found in step 2, where we start to build a controller named `PhoneListCtrl`. The example provided in the tutorial shows the following JavaScript:

```
var phonecatApp = angular.module('phonecatApp', []);
phonecatApp.controller('PhoneListCtrl', function ($scope)
{
    $scope.phones = [
        { 'name': 'Nexus S',
          'snippet': 'Fast just got faster with Nexus S.' },
```

```
    {'name': 'Motorola XOOM™ with Wi-Fi',
      'snippet': 'The Next, Next Generation tablet.'},
    {'name': 'MOTOROLA XOOM™',
      'snippet': 'The Next, Next Generation tablet.'}
  ];
});
```

This code snippet is typical of Angular JavaScript syntax. We start by creating a variable named `phonecatApp`, and register this as an Angular module by calling the `module` function on the `angular` global instance. The first argument to the `module` function is a global name for the Angular module, and the empty array is a place-holder for other modules that will be injected via Angular's Dependency Injection routines.

We then call the `controller` function on the newly created `phonecatApp` variable with two arguments. The first argument is the global name of the controller, and the second argument is a function that accepts a specially named Angular variable named `$scope`. Within this function, the code sets the `phones` object of the `$scope` variable to be an array of JSON objects, each with a `name` and `snippet` property.

If we continue reading through the tutorial, we find a unit test that shows how the `PhoneListCtrl` controller is used:

```
describe('PhoneListCtrl', function(){
  it('should create "phones" model with 3 phones', function() {
    var scope = {},
        ctrl = new PhoneListCtrl(scope);

    expect(scope.phones.length).toBe(3);
  });
});
```

The first two lines of this code snippet use a global function called `describe`, and within this function another function called `it`. These two functions are part of a unit testing framework named Jasmine. We will cover unit testing in our next chapter, but for the time being, let's focus on the rest of the code.

We declare a variable named `scope` to be an empty JavaScript object, and then a variable named `ctrl` that uses the `new` keyword to create an instance of our `PhoneListCtrl` class. The `new PhoneListCtrl(scope)` syntax shows that Angular is using the definition of the controller just like we would use a normal class in TypeScript.

Building the same object in TypeScript would allow us to use TypeScript classes, as follows:

```
var phonecatApp = angular.module('phonecatApp', []);

class PhoneListCtrl {
  constructor($scope) {
    $scope.phones = [
      { 'name': 'Nexus S',
        'snippet': 'Fast just got faster' },
      { 'name': 'Motorola',
        'snippet': 'Next generation tablet' },
      { 'name': 'Motorola Xoom',
        'snippet': 'Next, next generation tablet' }
    ];
  }
};
```

Our first line is the same as in our previous JavaScript sample. We then, however, use the TypeScript class syntax to create a class named `PhoneListCtrl`. By creating a TypeScript class, we can now use this class as shown in our Jasmine test code:

`ctrl = new PhoneListCtrl(scope)`. The constructor function of our `PhoneListCtrl` class now acts as the anonymous function seen in the original JavaScript sample:

```
phonecatApp.controller('PhoneListCtrl', function ($scope) {
  // this function is replaced by the constructor
})
```

Angular classes and \$scope

Let's expand our `PhoneListCtrl` class a little further, and have a look at what it would look like when completed:

```
class PhoneListCtrl {
  myScope: IScope;
  constructor($scope, $http: ng.IHttpService, Phone) {
    this.myScope = $scope;
    this.myScope.phones = Phone.query();
    $scope.orderProp = 'age';
    _.bindAll(this, 'GetPhonesSuccess');
  }
  GetPhonesSuccess(data: any) {
    this.myScope.phones = data;
  }
};
```

The first thing to note in this class, is that we are defining a variable named `myScope`, and storing the `$scope` argument that is passed in via the constructor, into this internal variable. This is again because of JavaScript's lexical scoping rules. Note the call to `_.bindAll` at the end of the constructor. This Underscore utility function will ensure that whenever the `GetPhonesSuccess` function is called, it will use the variable `this` in the context of the class instance, and not in the context of the calling code. We will discuss the usage of `_.bindAll` in detail in a later chapter.

The `GetPhonesSuccess` function uses the `this.myScope` variable within its implementation. This is why we needed to store the initial `$scope` argument in an internal variable.

Another thing we notice from this code, is that the `myScope` variable is typed to an interface named `IScope`, which will need to be defined as follows:

```
interface IScope {
    phones: IPhone[];
}
interface IPhone {
    age: number;
    id: string;
    imageUrl: string;
    name: string;
    snippet: string;
};
```

This `IScope` interface just contains an array of objects of type `IPhone` (pardon the unfortunate name of this interface – it can hold Android phones as well).

What this means is that we don't have a standard interface or TypeScript type to use when dealing with `$scope` objects. By its nature, the `$scope` argument will change its type depending on when and where the Angular runtime calls it, hence our need to define an `IScope` interface, and strongly type the `myScope` variable to this interface.

Another interesting thing to note on the constructor function of the `PhoneListCtrl` class is the type of the `$http` argument. It is set to be of type `ng.IHttpService`. This `IHttpService` interface is found in the declaration file for Angular. In order to use TypeScript with Angular variables such as `$scope` or `$http`, we need to find the matching interface within our declaration file, before we can use any of the Angular functions available on these variables.

The last point to note in this constructor code is the final argument, named `Phone`. It does not have a TypeScript type assigned to it, and so automatically becomes of type `any`. Let's take a quick look at the implementation of this `Phone` service, which is as follows:

```
var phonecatServices =
  angular.module('phonecatServices', ['ngResource']);

phonecatServices.factory('Phone',
  [
    '$resource', ($resource) => {
      return $resource('phones/:phoneId.json', {}, {
        query: {
          method: 'GET',
          params: {
            phoneId: 'phones'
          },
          isArray: true
        }
      });
    }
  ]
);
```

The first line of this code snippet again creates a global variable named `phonecatServices`, using the `angular.module` global function. We then call the factory function available on the `phonecatServices` variable, in order to define our `Phone` resource. This factory function uses a string named `'Phone'` to define the `Phone` resource, and then uses Angular's dependency injection syntax to inject a `$resource` object. Looking through this code, we can see that we cannot easily create standard TypeScript classes for Angular to use here. Nor can we use standard TypeScript interfaces or inheritance on this Angular service.

Angular TypeScript compatibility

When writing Angular code with TypeScript, we are able to use classes in certain instances, but must rely on the underlying Angular functions such as `module` and `factory` to define our objects in other cases. Also, when using standard Angular services, such as `$http` or `$resource`, we will need to specify the matching declaration file interface in order to use these services. We can therefore describe the Angular library as having medium compatibility with TypeScript.

Inheritance – Angular versus Backbone

Inheritance is a very powerful feature of object-oriented programming, and is also a fundamental concept when using JavaScript frameworks. Using a Backbone controller or an Angular controller within each framework relies on certain characteristics, or functions being available. We have seen, however, that each framework implements inheritance in a different way.

As JavaScript does not have the concept of inheritance, each framework needs to find a way to implement it, so that the framework can allow us to extend base classes and their functionality. In Backbone, this inheritance implementation is via the `extend` function of each Backbone object. As we have seen, the TypeScript `extends` keyword follows a similar implementation to Backbone, allowing the framework and language to dovetail each other.

Angular, on the other hand, uses its own implementation of inheritance, and defines functions on the angular global namespace to create classes (that is `angular.module`). We can also sometimes use the instance of an application (that is `<appName>.controller`) to create modules or controllers. We have found, though, that Angular uses controllers in a very similar way to TypeScript classes, and we can therefore simply create standard TypeScript classes that will work within an Angular application.

So far, we have only skimmed the surface of both the Angular TypeScript syntax and the Backbone TypeScript syntax. The point of this exercise was to try and understand how TypeScript can be used within each of these two third party frameworks.

Be sure to visit <http://todomvc.com>, and have a look at the full source-code for the Todo application written in TypeScript for both Angular and Backbone. They can be found on the **Compile-to-JS** tab in the example section. These running code samples, combined with the documentation on each of these sites, will prove to be an invaluable resource when trying to write TypeScript syntax with an external third party library such as Angular or Backbone.

Angular 2.0

The Microsoft TypeScript team and the Google Angular team have just completed a months long partnership, and have announced that the upcoming release of Angular, named Angular 2.0, will be built using TypeScript. Originally, Angular 2.0 was going to use a new language named AtScript for Angular development. During the collaboration work between the Microsoft and Google teams, however, the features of AtScript that were needed for Angular 2.0 development have now been implemented within TypeScript. This means that the Angular 2.0 library will be classed as highly compatible with TypeScript, once the Angular 2.0 library, and the 1.5 edition of the TypeScript compiler are available.

ExtJs

ExtJs is a popular JavaScript library that has a wide variety of widgets, grids, graphing components, layout components and more. With release 4.0, ExtJs incorporated a model, view, controller style of application architecture into their libraries. Although it is free for open-source development, ExtJs requires a license for commercial use. It is popular with development teams that are building web-enabled desktop replacements, as its look and feel is comparable to normal desktop applications. ExtJs, by default, ensures that each application or component will look and feel exactly the same, no matter which browser it is run in, and it requires little or no need for CSS or HTML.

The ExtJs team, however, has not released an official TypeScript declaration file for ExtJs, despite much community pressure. Thankfully, the wider JavaScript community has come to the rescue, beginning with Mike Aubury. He wrote a small utility program to generate declaration files from the ExtJs documentation (<https://github.com/zz9pa/extjsTypescript>).

Whether this work influenced the current version of the ExtJs definitions on DefinitelyTyped or not, remains to be seen, but the original definitions from Mike Aubury and the current version from brian428 on DefinitelyTyped are very similar.

Creating classes in ExtJs

ExtJs is a JavaScript library that does things in its own way. If we were to categorize Backbone, Angular and ExtJs, we might say that Backbone is a highly compliant TypeScript library. In other words, the language features of classes and inheritance within TypeScript are highly compliant with Backbone.

Angular in this case would be a partially compliant library, with some elements of Angular objects complying with the TypeScript language features. ExtJs, on the other hand, would be a minimally compliant library, with little or no TypeScript language features applicable to the library.

Let's take a look at a sample ExtJs 4.0 application written in TypeScript. Consider the following code:

```
Ext.application(  
    {  
        name: 'SampleApp',  
        appFolder: '/code/sample',  
        controllers: ['SampleController'],  
        launch: () => {  
  
            Ext.create('Ext.container.Viewport', {  
                layout: 'fit',  
                items: [{  
                    xtype: 'panel',  
                    title: 'Sample App',  
                    html: 'This is a Sample Viewport'  
                }]  
            });  
  
        }  
    }  
);
```

We start by creating an ExtJs application by calling the `application` function on the `Ext` global instance. The `application` function then uses a JavaScript object, enclosed within the first and last curly braces `{ }` to define properties and functions. This ExtJs application sets the `name` property to be `SampleApp`, the `appFolder` property to be `/code/sample`, and the `controllers` property to be an array with a single entry: `'SampleController'`.

We then define a `launch` property, which is an anonymous function. This launch function then uses the `create` function on the global `Ext` instance to create a class. The `create` function uses the `"Ext.container.Viewport"` name to create an instance of the `Ext.container.Viewport` class, which has the properties `layout` and `items`. The `layout` property can only contain one of a specific set of values, for example `'fit'`, `'auto'` or `'table'`. The `items` array contains further ExtJs specific objects, which are created depending on what their `xtype` property suggests.

ExtJs is one of those libraries that is not intuitive. As a programmer, you will need to have one browser window open with the library documentation at all times, and use it to figure out what each property means for each type of available class. It also has a lot of magic strings – in the preceding sample, the `Ext.create` function would fail if we miss-typed the `'Ext.container.Viewport'` string, or simply forgot to capitalize it in the right places. To ExtJs, `'viewport'` is different to `'ViewPort'`. Remember that one of our solutions to magic strings within TypeScript is to use enums. Unfortunately, the current version of the ExtJs declaration file does not have a set of enums for these class types.

Using type casting

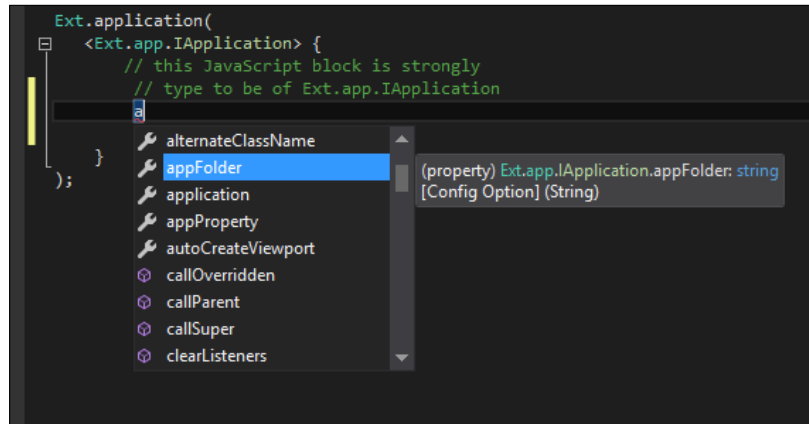
We can, however, use the TypeScript language feature of type casting to help with writing ExtJs code. If we know what type of ExtJs object we are trying to create, we can cast the JavaScript object to this type, and then use TypeScript to check whether the properties we are using are correct for that type of ExtJs object. To help with this concept, let's just take the outer definition of the `Ext.application` into account. Stripped of the inner code, the call to the `application` function on the `Ext` global object would be reduced to this:

```
Ext.application(  
  {  
    // properties of an Ext.application  
    // are set within this JavaScript  
    // object block  
  }  
);
```

Using the TypeScript declaration files, type casting, and a healthy dose of ExtJs documentation, we know that the inner JavaScript object should be of type `Ext.app.IApplication`, and we can therefore cast this object as follows:

```
Ext.application(  
  <Ext.app.IApplication> {  
    // this JavaScript block is strongly  
    // type to be of Ext.app.IApplication  
  }  
);
```

The second line of this code snippet now uses the TypeScript type casting syntax, to cast the JavaScript object between the curly braces { } to a type of `Ext.app.IApplication`. This gives us strong type checking, and Intellisense, as shown in the following screenshot:



Visual Studio intellisense for an ExtJS configuration block

In a similar manner, these explicit type casts can be used on any JavaScript object that is being used to create ExtJS classes. The declaration file for ExtJS currently on DefinitelyTyped uses the same names for its object definitions as the ExtJS documentation uses, so finding the correct type should be rather simple.

The preceding technique of using explicit type casting is just about the only language feature of TypeScript that we can use with the ExtJS library – but this still highlights how strong typing of objects can assist us in our development experience, making our code more robust and resistant to errors.

ExtJS specific TypeScript compiler

If you are using ExtJS on a regular basis, then you may want to take a look at the work done by Gareth Smith, Fabio Parra dos Santos and their team at <https://github.com/fabioparra/TypeScript>. This project is a fork of the TypeScript compiler that will emit ExtJS classes from standard TypeScript classes. Using this version of the compiler turns the tables on normal ExtJS development, allowing for natural TypeScript class syntax, the use of inheritance via the `extends` keyword, as well as natural module naming, without the need for magic strings. The work done by this team shows that because the TypeScript compiler is open-source, it can be extended and modified to emit JavaScript in a specific way, or to target a specific library. Hats off to Gareth, Fabio and their team for their ground-breaking work in this area.

Summary

In this chapter, we have had a look at third party JavaScript libraries and how they can be used within a TypeScript application. We started by looking at the various ways of including community released versions of TypeScript declaration files within our projects, from downloading the raw files, to using package managers like NuGet and TSD. We then looked at three types of third party libraries, and discussed how to integrate these libraries with TypeScript. We explored Backbone, which can be categorized as a highly compliant third party library, Angular, which is a partially compliant library, and ExtJs which is a minimally compliant library. We saw how various features of the TypeScript language can co-exist with these libraries, and showed what TypeScript equivalent code would look like in each of these cases. In the next chapter, we will look at Test Driven Development, and explore some of the libraries that are available for unit testing, integration testing, and automated acceptance testing.

6

Test Driven Development

Over the past few years, the popularity of the **Model View Controller (MVC)**, **Model View Presenter (MVP)**, and **Model View ViewModel (MVVM)** patterns has given rise to a range of third-party JavaScript libraries, each implementing their own version of these patterns. Backbone, for example, could be described as an MVP implementation, where the view acts as a presenter. ExtJS 4 introduced an MVC pattern to their framework, and Angular could be described as more of an MVVM framework. When discussing this group of patterns together, they have been described by some as **Model View Whatever (MVW)**, or **Model View Something (MV*)**.

Some of the benefits of this MV* style of writing applications include modularity and separation of concerns. This MV* style of building applications also brings with it a huge advantage—the ability to write testable JavaScript. Using MV* allows us to unit test, integration test, and function test almost all of our beautifully hand-crafted JavaScript. This means that we can test our rendering functions to ensure that DOM elements are correctly shown on the page. We can also simulate button clicks, drop-down selections, and animations. We can also extend these tests to page transitions, including login pages and home pages. By building a large set of tests for our application, we will gain confidence that our code works as expected, and it will allow us to refactor our code at any time.

In this chapter, we will look at Test Driven Development in relation to TypeScript. We will discuss some of the more popular testing frameworks, write some unit tests, and then discuss test runners and continuous integration techniques.

Test Driven Development

Test Driven Development (TDD) is a development process, or a development paradigm, that starts with tests and drives the momentum of a piece of production code through these tests. Test Driven Development means asking the question "how do I know that I have solved the problem?" instead of just "how do I solve the problem?"

The basic steps of a test driven approach are the following:

- Write a test that fails
- Run the test to ensure that it fails
- Write the code to make the test pass
- Run the test to see that it passes
- Run all tests to see that the new code does not break any others
- Repeat the process

Using Test Driven Development practices is really a mindset. Some developers follow this approach and write tests first, while others write their code first and their tests afterwards. Then there are some that don't write tests at all. If you fall into the last category, then hopefully, the techniques you learn in this chapter will help you to get started in the right direction.

There are so many excuses out there for not writing unit tests. Some typical excuses include phrases like "the test framework was not in our original quote", or "it will add 20 percent to the development time", or "the tests are outdated so we don't run them anymore". The truth is, though, that in this day and age, we cannot afford not to write tests. Applications grow in size and complexity, and requirements change over time. An application that has a good suite of tests can be modified far more quickly, and will be much more resilient to future requirement changes than one that does not have tests. This is when the real cost savings of unit testing become apparent. By writing unit tests for your application, you are future-proofing it, and ensuring that any change to the code base does not break existing functionality.

TDD in the JavaScript space adds another layer to our code coverage. Quite often, development teams will write tests that target only the server-side logic of an application. As an example, in the Visual Studio space, these tests are often written to only target the MVC framework of controllers, views, and underlying business logic. It has always been fairly difficult to test the client-side logic of an application—in other words, the actual rendered HTML and user-based interactions.

JavaScript testing frameworks provide us with tools to fill this gap. We can now start to unit test our rendered HTML, as well as simulate user interactions such as filling in forms and clicking on buttons. This extra layer of testing, combined with server-side testing, means that we have a way to unit testing each layer of our application—from server-side business logic, through server-side page rendering, right through to rendering and user interactions. The ability to unit test frontend user interactions is one of the greatest strengths of any JavaScript MV* framework. In fact, it could even influence the architectural decisions you make when choosing a technology stack.

Unit, integration and acceptance tests

Automated tests can be broken up into three general areas, or types of tests—unit tests, integration tests, and acceptance tests. We can also describe these tests as either black box or white box tests. White box tests are tests where the internal logic or structure of the code under test is known to the tester. Black box tests, on the other hand, are tests where the internal design and or logic are not known to the tester.

Unit tests

A unit test is typically a white box test where all of the external interfaces to a block of code are mocked or stubbed out. If we are testing some code that does an asynchronous call to load a block of JSON for example, unit testing this code would require mocking out the returned JSON. This technique ensures that the object under test is always given a known set of data. When new requirements come along, this known set of data can grow and expand, of course. Objects under test should be designed to interact with interfaces so that those interfaces can be easily mocked or stubbed out in a unit test scenario.

Integration tests

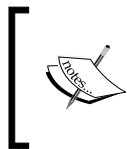
Integration tests are another form of white box tests that allow the object under test to run in an environment close to how it would in real code. In our preceding example, where some code does an asynchronous call to load a block of JSON, an integration test would need to actually call the **Representational State Transfer (REST)** services that generate the JSON. If this REST service relied upon data from a database, then the integration test would need data in the database that matched the integration test scenario. If we were to describe a unit test as having a boundary around the object under test, then an integration test is simply an expansion of this boundary to include dependent objects or services.

Building automated integration tests for your applications will improve the quality of your application immensely. Consider the case in the scenario that we have been using – where a block of code calls a REST service for some JSON data. Someone could easily change the structure of the JSON data that the REST service returns. Our unit tests will still pass, as they are not actually calling the REST server-side code, but our application will be broken because the returned JSON is not what we are expecting.

Without integration tests, these types of errors will only be picked up in later stages of manual testing. Thinking about integration tests, implementing specific data sets for integration tests, and building them into your test suite will eliminate these sorts of bugs early.

Acceptance tests

Acceptance tests are black box tests, and are generally scenario-based. They may incorporate multiple user screens or user interactions in order to pass. These tests are also generally carried out by the testing team, as it may require logging in to the application, searching for a particular set of data, updating the data, and so on. With some planning, we can also automate parts of these acceptance tests into an integration suite, as we have the ability in JavaScript to find and click buttons, insert data into required fields, or select drop-down items. The more acceptance tests a project has, the more robust it will be.



In the Test Driven Development methodology, every bug that is picked up by a manual testing team must result in the creation of new unit, integration, or acceptance tests. This methodology will help to ensure that once a bug is found and fixed, it never reappears again.

Using continuous integration

When writing unit tests for any application, it quickly becomes important to set up a build server and run your tests as part of each source control check in. When your development team grows beyond a single developer, using a **Continuous Integration (CI)** build server becomes imperative. This build server will ensure that any code committed to the source control server passes all known unit tests, integration tests, and automated acceptance tests. The build server is also responsible for labeling a build and generating any deployment artifacts that need to be used during deployment.

The basic steps of a build server would be as follows:

- Check out the latest version of the source code, and increase the build number
- Compile the application on the build server
- Run any server-side unit tests
- Package the application for deployment
- Deploy the package to a build environment
- Run any server-side integration tests
- Run any JavaScript unit, integration, and acceptance tests
- Mark the change set and build number as passed or failed
- If the build failed, notify those responsible for breaking it



The build server should fail if any one of the preceding steps fail.

Benefits of continuous integration

Using a build server to run through the preceding steps brings huge benefits to any development team. Firstly, the application is compiled on the build server – which means that any tools or external libraries used, will need to be installed on the build server. This gives your development team the opportunity to document exactly what software needs to be installed on a new machine in order to compile or run your application.

Secondly, a standard set of server-side unit tests can be run before the packaging step is attempted. In a Visual Studio project, these would be C# unit tests built with any of the popular .NET testing frameworks, such as MSTest, NUnit, or xUnit.

Next, the entire application's packaging step is run. Let's assume for a moment that a developer has included a new JavaScript library within the project, but forgotten to add it to the Visual Studio solution. In this case, all of the tests will run on their local computer, but will break the build because of a missing library file. If we were to deploy the site at this stage, running the application would result in a 404 error – file not found. By running a packaging step, these sort of errors are quickly found.

Once a successful packaging step has been completed, the build server should deploy the site to a specially marked build environment. This build environment is only used for CI builds, and must therefore have its own database instances, web service references, and so on, set up specifically for CI builds. Again, actually doing a deployment to a target environment tests the deployment artifacts, as well as the deployment process. By setting up a build environment for automated package deployment, your team is again able to document the requirements and process for deployment.

At this stage, we have a full instance of our website up and running on an isolated build environment. We can then easily target specific web pages that will run our JavaScript tests, and also run integration or automated acceptance tests – directly on the full version of the website. In this way, we can write tests that target the real-life website REST services, without having to mock out these integration points. So in effect, we are testing the application from the ground up. Obviously, we may need to ensure that our build environment has a specific set of data that can be used for integration testing, or a way of generating the required data sets that our integration tests will need.

Selecting a build server

There are a number of continuous integration build servers out there, including TeamCity, Jenkins, and **Team Foundation Server (TFS)**.

Team Foundation Server

TFS needs a specific configuration on its build agents to be able to run instances of a web browser. With larger projects, actually running the JavaScript tests within a specific browser makes sense, and soon becomes a required step. You may need to support more than one browser, and want to run your tests within Firefox, Chrome, IE, Safari, or others. TFS also uses **Windows Workflow Foundation (WF)** to configure build steps, which takes a fair amount of experience and knowledge to modify.

Jenkins

Jenkins is an open source, free-to-use CI build server. It has wide community usage, and many plugins. Installation and configuration of Jenkins is fairly straightforward, and Jenkins will allow processes to run browser instances, making it compatible with browser-based JavaScript unit tests. Jenkins build steps are command-line-based, and it sometimes takes a little nous to configure build steps correctly.

TeamCity

A very popular, and very powerful, build server that is free to set up is TeamCity. TeamCity allows free installation if you have a small number of developers (< 20), and a small number of projects (< 20). A full commercial license is only around \$1,500.00, which makes it affordable for most organizations. Configuring build steps in TeamCity is much easier than in Jenkins or TFS, as it uses a wizard style of configuration depending on the type of build step you are creating. TeamCity also has a rich set of functionality around unit tests, with the ability to show graphs per unit test, and is therefore considered best of breed for build servers.

Unit testing frameworks

There are many JavaScript unit testing frameworks available, and also a few that have been written in TypeScript. Two of the most popular JavaScript frameworks are Jasmine (<http://jasmine.github.io/>) and QUnit (<http://qunitjs.com/>). If you are writing Node TypeScript code, then you might want to have a look at mocha (<https://github.com/mochajs/mocha/wiki>).

Two of the TypeScript-based testing frameworks are MaxUnit (<https://github.com/KnowledgeLakegithub/MaxUnit>) and tsUnit (<https://github.com/SteveFenton/tsUnit>). Unfortunately, both MaxUnit and tsUnit are newcomers in this space, and therefore may not have the features that are inherent in the older, more popular frameworks. MaxUnit, for example, did not have any documentation at the time of writing, and tsUnit does not have a test reporting framework compatible with CI build servers. Over time, these TypeScript frameworks may grow, but seeing how easy it is to work with third-party libraries and use DefinitelyTyped declaration files, writing unit tests for either QUnit or Jasmine becomes a very simple process.

For the rest of this chapter, we will be using Jasmine 2.0 as our testing framework.

Jasmine

For this section of the chapter, we will create a new Visual Studio project that is based on the MVC framework project type. For now, we can just use the empty MVC template.

Jasmine can be installed into our new TypeScript project with the following two NuGet packages:

```
Install-Package JasmineTest
```

```
Install-Package jasmine.TypeScript.DefinitelyTyped
```


With these two packages in place, we have the required JavaScript libraries and TypeScript definition files in place to begin writing Jasmine tests.



The default installation from NuGet for `JasmineTest` uses the ASP.NET MVC framework, and creates a `JasmineController` in the `Controllers` directory. If you are not using the MVC framework, or are installing this package in a Node environment, then this `JasmineController` should be deleted, as it will cause compilation errors. Later in this chapter, we will show how to run integration tests against this `JasmineController`, so it's best to leave it in place for the time being.

A simple Jasmine test

Jasmine uses a simple format for writing tests. Consider the following TypeScript code:

```
describe("tests/01_SimpleJasmineTests.ts ", () => {
  it("should fail", () => {
    var undefinedValue;
    expect(undefinedValue).toBeDefined();
  });
});
```

This snippet starts with a Jasmine function called `describe`, which takes two arguments. The first argument is the name of the test suite, and the second is an anonymous function that contains our test suite. The next line uses the Jasmine function named `it`, which also takes two arguments. The first argument is the test name, and the second argument is an anonymous function that contains our test; in other words, whatever is within the `it` anonymous function is our actual test. This test starts by defining a variable, named `undefinedValue`, but does not actually set its value. Next, we use the Jasmine function `expect`. Just by reading the code of this `expect` statement, we can quickly understand what the unit test is doing. It is expecting that the value of the `undefinedValue` variable should be defined, that is, not `undefined`.

The `expect` function takes a single argument, and returns a Jasmine matcher. We can then call any of the Jasmine matcher functions to assess the value passed into `expect` against the matcher function. The `expect` keyword is similar to the `Assert` keyword in other testing libraries. The format of the `expect` statements are human-readable, making Jasmine expectations relatively simple to understand.

Jasmine SpecRunner.html file

In order to run this test, we will need an HTML page that includes all the relevant Jasmine third-party libraries, as well as our test JavaScript file. We can create a `SpecRunner.html` file with the following HTML within it:

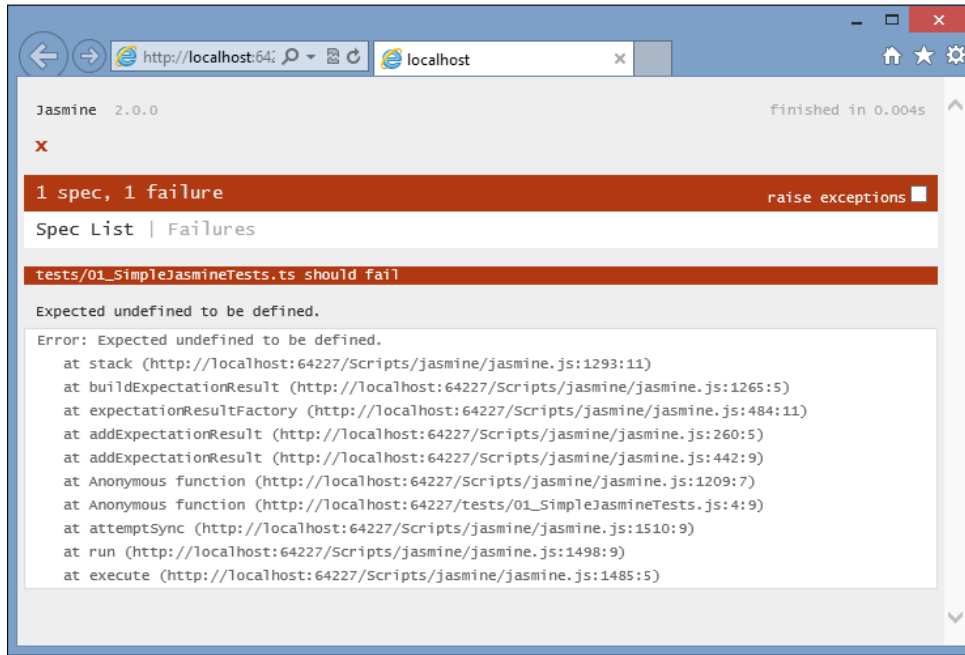
```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head>
    <title>Jasmine Spec Runner</title>
    <link rel="shortcut icon" type="image/png"
          href="/Content/jasmine/jasmine_favicon.png">
    <link rel="stylesheet" type="text/css"
          href="/Content/jasmine/jasmine.css">
    <script type="text/javascript"
            src="/Scripts/jasmine/jasmine.js"></script>
    <script type="text/javascript"
            src="/Scripts/jasmine/jasmine-html.js"></script>
    <script type="text/javascript"
            src="/Scripts/jasmine/boot.js"></script>
    <script type="text/javascript"
            src="/tests/01_SimpleJasmineTests.js"></script>

  </head>
  <body>

</body>
</html>
```

This HTML page is simply including the required Jasmine files, `jasmine.css`, `jasmine.js`, `jasmine-html.js`, and `boot.js`. The last line includes the compiled JavaScript file from our TypeScript test file.

If we set this page as our startup page within Visual Studio and run it, we should see one failing unit test:



SpecRunner.html page showing Jasmine output

Excellent! We are following the test-driven development process by firstly creating a failing unit test. The results are exactly what we expect. Our variable named `undefinedVariable` has not yet had a value assigned to it, and therefore will be `undefined`. If we follow the next step of the TDD process, we should write the code that makes the test pass. Updating our test as follows will ensure that the test will pass:

```
describe("tests/01_SimpleJasmineTests.ts ", () => {
  it("value that has been assigned should be defined", () => {
    var undefinedValue = "test";
    expect(undefinedValue).toBeDefined();
  });
});
```

Note that we have updated our test name to describe what the test is trying to accomplish. To make the test pass, we are simply assigning the value `"test"` to our `undefinedValue` variable. Running the `SpecRunner.html` page now will show a passing test.

Matchers

Jasmine has a wide range of matchers that can be used within tests, and also allows us to write and include custom matchers. The syntax of Jasmine matchers is pretty self-explanatory, as can be seen from the following TypeScript code:

```
var undefValue;  
expect(undefValue).not.toBeDefined();
```

Here, we are using the `.not` matcher syntax to check that the variable `undefValue` is indeed undefined.

```
var definedValue = 2;  
expect(definedValue).not.toBe(null);
```

This `expect` statement uses the `not.toBe` matcher to ensure that the `definedValue` variable is not null.

```
expect(definedValue).toBe(2);
```

Here, we are using the `.toBe` matcher to check that the `definedValue` is in fact a number with the value of two.

```
expect(definedValue.toString()).toEqual("2");
```

This `expect` statement is using the `toEqual` matcher to ensure that the `toString` function will return the string value of "2".

```
var trueValue = true;  
expect(trueValue).toBeTruthy();  
expect(trueValue).not.toBeFalsy();
```

Here, we are testing for boolean values, using the `toBeTruthy` and `toBeFalsy` matchers.

```
var stringValue = "this is a string";  
expect(stringValue).toContain("is");  
expect(stringValue).not.toContain("test");
```

Finally, we can also use the `toContain` matcher to parse a string, and test whether it contains another string—or use the `.not` matcher with `toContain` for the reverse test.

Be sure to head over to the Jasmine website for a full list of matchers, as well as details on writing your own custom matchers.

Test startup and teardown

As in other testing frameworks, Jasmine provides a mechanism to define functions that will run before and after each test, or as a test startup and teardown mechanism. In Jasmine, the `beforeEach` and `afterEach` functions act as test startup and teardown functions, as can be seen from the following TypeScript code:

```
describe("beforeEach and afterEach tests", () => {
    var myString;

    beforeEach(() => {
        myString = "this is a test string";
    });
    afterEach(() => {
        expect(myString).toBeUndefined();
    });

    it("should find then clear the myString variable", () => {
        expect(myString).toEqual("this is a test string");
        myString = undefined;
    });
});
```

In this test, we define a variable named `myString`, at the start of the anonymous function. As we know from JavaScript lexical scoping rules, this `myString` variable will then be available for use within each of the following `beforeEach`, `afterEach`, and `it` functions. Within the `beforeEach` function, this variable is set to a string value. Within the `afterEach` function, the variable is tested to see that it has been reset to `undefined`. Our expectation within our test checks is that this variable has been set via the `beforeEach` function. At the end of our test, we then reset the variable to be `undefined`. Note that the `afterEach` function is also calling an `expect` – in this case to ensure that the test has reset the variable back to `undefined`.



The Jasmine 2.1 version introduces a second version of setup and teardown, called `beforeAll` and `afterAll`. At the time of writing this book, though, the versions of both the `jasmine.js` and `jasmine.d.ts` files available from NuGet had not been updated to v2.1.

Data-driven tests

To show how extensible the Jasmine testing library is, JP Castro wrote a very short but powerful utility to provide data-driven tests within Jasmine. His blog on this topic can be found here (<http://blog.jphpsf.com/2012/08/30/drying-up-your-javascript-jasmine-tests/>), and the GitHub repository can be found here (<https://github.com/jphpsf/jasmine-data-provider>). This simple extension allows us to write intuitive Jasmine tests that take a parameter as part of each test, as follows:

```
describe("data driven tests", () => {
  using<string>("valid values", [
    "first string",
    "second string",
    "third string"
  ], (value) => {
    it("should contain string (" + value + ")", () => {
      expect(value).toContain("string");
    });
  });
});
```

Here, we are wrapping our `it` test function within another function called `using`. This `using` function takes three parameters: a string description of the value set, an array of values, and a function definition. This last function definition uses the variable `value`, and will invoke our test using this value. Note also in the call to our test, we are changing the test name on the fly, to include the `value` parameter that is passed in. This is necessary in order for each test to have a unique test name.

The preceding solution just needs JP Castro's Jasmine extension, shown in the following JavaScript code:

```
function using(name, values, func) {
  for (var i = 0, count = values.length; i < count; i++) {
    if (Object.prototype.toString.call(values[i])
        !== '[object Array]')
    {
      values[i] = [values[i]];
    }
    func.apply(this, values[i]);
  }
}
```

This is a very simple function named `using`, that takes the three parameters that we mentioned earlier. The function does a simple loop through the array values, and passes in each array value to our test.

The last item that we will need is a TypeScript definition file for the preceding `using` function. This is a very simple function declaration as follows:

```
declare function using<T>(
  name: string,
  values : T [],
  func : (T) => void
);
```

This TypeScript declaration uses the generic syntax `<T>` to ensure that the same type is used for both the second and third arguments. With this declaration in place, and the JavaScript `using` function, our code will compile correctly, and the tests will run through once for each value in the data array:

```
data driven tests
should contain string (first string)
should contain string (second string)
should contain string (third string)
```

Using spies

Jasmine also has a very powerful feature that allows your tests to see if a particular function was called, and what parameters it was called with. It can also be used to create mocks and stubs. All of this functionality is rolled into what Jasmine calls spies.

Consider the following test:

```
class MySpiedClass {
  testFunction(arg1: string) {
    console.log(arg1);
  }
}

describe("simple spy", () => {
  it("should register a function call", () => {
    var classInstance = new MySpiedClass();
    spyOn(classInstance, 'testFunction');

    classInstance.testFunction("test");
  });
});
```


```

        expect(classInstance.testFunction).toHaveBeenCalled();
    });
});

```

We start with a simple class named `MySpiedClass`, that has a single function `testFunction`. This function takes a single argument, and logs the argument to the console.

Our test starts by creating a new instance of the `MySpiedClass`, and assigns it to a variable named `classInstance`. We then create a Jasmine spy on the function `testFunction` of the `classInstance` variable. Once we have a spy created, we can call the function. Our expectation then checks whether the function was called. This is the essence of a spy. Jasmine will "watch" the `testFunction` function of the instance of `MySpiedClass` to see whether or not it was called.



Jasmine spies, by default, block the call to the underlying function. In other words, they replace the function you are trying to call with a Jasmine delegate. If you need to spy on a function, but still need the body of the function to execute, you must specify this behavior using the `.and.callThrough()` fluent syntax.

While this is a very trivial example, spies become very powerful in a number of different testing scenarios. Classes or functions that take callback parameters, for example, would need a spy to ensure that the callback function was in fact invoked.

Let's see how we can test that a callback function was invoked correctly. Consider the following TypeScript code:

```

class CallbackClass {
    doCallback(id: number, callback: (result: string) => void ) {
        var callbackValue = "id:" + id.toString();
        callback(callbackValue);
    }
}

class DoCallback {
    logValue(value: string) {
        console.log(value);
    }
}

```

In this code snippet, we define a class named `CallbackClass` that has a single function `doCallback`. This `doCallback` function takes an `id` argument of type `number`, and also a `callback` function. The `callback` function takes a `string` as an argument, and returns `void`.

The second class that we have defined has a single function named `logValue`. This function signature matches the callback function signature required on the `doCallback` function. Using Jasmine spies, we can test the logic of the `doCallback` function. This logic creates a string based on the `id` argument that was passed in, and then invokes the `callback` function with this string. Our test will need to ensure that this string is formatted correctly. Our Jasmine test, then, for this can be written as follows:

```
describe("using callback spies", () => {
  it("should execute callback with the correct string value", () =>
  {
    var doCallback = new DoCallback();
    var classUnderTest = new CallbackClass();

    spyOn(doCallback, 'logValue');
    classUnderTest.doCallback(1, doCallback.logValue);

    expect(callbackSpy.logValue).toHaveBeenCalled();
    expect(callbackSpy.logValue).toHaveBeenCalledWith("id:1");

  });
});
```

This test code firstly creates an instance of the class `CallbackClass`, and also an instance of the class `DoCallback`. We then create a spy on the `logValue` function of the `DoCallback` class. We then call the `doCallback` function, passing in a value of 1 as the first argument, and the `logValue` function as the second argument. Our expect statements on the last two lines check that the callback function `logValue` was actually called, and also what parameters it was called with.

Using spies as fakes

Another benefit of Jasmine spies is that they can act as fakes. In other words, instead of calling a real function, the call is delegated to the Jasmine spy. Jasmine also allows spies to return values – which can be useful when generating small mocking frameworks. Consider the following tests:

```
Class ClassToFake {
  getValue(): number {
    return 2;
  }
}
```

```
describe("using fakes", () => {
  it("calls fake instead of real function", () => {
    var classToFake = new ClassToFake();
    spyOn(classToFake, 'getValue')
      .and.callFake(() => { return 5; }
    );
    expect(classToFake.getValue()).toBe(5);
  });
});
```

We start with a class named `ClassToFake` that has a single function `getValue`, which returns 2. Our test then creates an instance of this class. We then call the Jasmine `spyOn` function to create a spy on the `getValue` function, and then use the `.and.callFake` syntax to attach an anonymous function as a fake function. This fake function will return 5 instead of the original `getValue` function that would have returned 2. The test then checks to see that when we call the `getValue` function on the `ClassToFake` instance, Jasmine will substitute our new fake function for the original `getValue` function, and return 5 instead of 2.

There are a number of variants of the Jasmine fake syntax, including methods to throw errors, or return values—again, please consult the Jasmine documentation for a full list of its faking capabilities.

Asynchronous tests

The asynchronous nature of JavaScript—made popular by AJAX and jQuery has always been one of the drawcards of the language, and is the principle architecture behind Node based applications. Let's have a quick look at an asynchronous class, and then describe how we should go about testing it. Consider the following TypeScript code:

```
class MockAsyncClass {
  executeSlowFunction(success: (value: string) => void) {
    setTimeout(() => {
      success("success");
    }, 1000);
  }
}
```

The `MockAsyncClass` has a single function named `executeSlowFunction`, which takes a function callback named `success`. Within the `executeSlowFunction` code, we are simulating an asynchronous call by using the `setTimeout` function, and only calling the `success` callback after 1000 milliseconds (1 second). This behavior is simulating a standard AJAX call (which would use both a `success` and an `error` callback), which could take a number of seconds to return – depending on the speed of the backend server, or the size of the data packet.

Our test for this `executeSlowFunction` may look as follows:

```
describe("asynchronous tests", () => {
  it("failing test", () => {

    var mockAsync = new MockAsyncClass();
    var returnedValue;
    mockAsync.executeSlowFunction((value: string) => {
      returnedValue = value;
    });
    expect(returnedValue).toEqual("success");
  });
});
```

Firstly, we instantiate an instance of the `MockAsyncClass`, and define a variable named `returnedValue`. We then call the `executeSlowFunction` with an anonymous function for the `success` callback function. This anonymous function sets the value of `returnedValue` to whatever value was passed in from the `MockAsyncClass`. Our expectation is that the `returnedValue` should equal `"success"`. If we run this test now, however, our test will fail with the following error message:

Expected undefined to equal 'success'.

What is happening here, is that because the `executeSlowFunction` is asynchronous, JavaScript will not wait until the callback function is called, before executing the next line of code. This means that the expectation is being called before the `executeSlowFunction` has had a chance to call our anonymous callback function (setting the value of `returnedValue`). If you put a breakpoint on the `expect(returnedValue).toEqual("success")` line, and another breakpoint on the `returnedValue = value` line, you will see that the `expect` line is called first, and the `returnedValue` line is only called after a second. This timing issue is what is causing this test to fail. We need to somehow have our test wait until the `executeSlowFunction` has invoked the callback, before we execute our expectations.

Using the done() function

Jasmine version 2.0 has introduced a new syntax to help us with these sort of asynchronous tests. In any `beforeEach`, `afterEach`, or `it` function, we pass an argument named `done`, which is a function, and then invoke it at the end of our asynchronous code. Consider the following test:

```
describe("asynch tests with done", () => {
  var returnedValue;

  beforeEach((done) => {
    returnedValue = "no_return_value";
    var mockAsync = new MockAsyncClass();
    mockAsync.executeSlowFunction((value: string) => {
      returnedValue = value;
      done();
    });
  });

  it("should return success after 1 second", (done) => {
    expect(returnedValue).toEqual("success");
    done();
  });
});
```

Firstly, we have moved the `returnedValue` variable outside of our test, and have included a `beforeEach` function to run before our actual test. This `beforeEach` function firstly resets the value of `returnValue`, and then sets up the `MockAsyncClass` instance. Finally it calls the `executeSlowFunction` on this instance.

Note how the `beforeEach` function takes a parameter named `done`, and then calls this `done` function after the `returnedValue = value` line has been called. Notice too, that the second parameter to the `it` function now also takes a `done` parameter, and invokes this `done` function when the test is finished.




From the Jasmine documentation: The spec will not start until the `done` function is invoked in the call to `beforeEach`, and the spec will not complete until the `done` function is called. By default, Jasmine will wait for 5 seconds before causing a timeout failure. This can be overridden using the `jasmine.DEFAULT_TIMEOUT_INTERVAL` variable.

Jasmine fixtures

Many times, our code is responsible for either reading in, or in most cases manipulating DOM elements from JavaScript. This means that any running code that relies on a DOM element could fail, if the underlying HTML does not contain the correct element or group of elements. Another Jasmine extension library named `jasmine-jquery` allows us to inject HTML elements into the DOM before our tests execute, and will remove them from the DOM after the test is run.

At the time of writing this book, this library was not available on NuGet, so we will need to download the `jasmine-jquery.js` file the old-fashioned way, and include it in our project. The TypeScript definition file is, however, available on NuGet:

Install-package Jasmine-jquery.TypeScript.DefinitelyTyped

 We will need to also update the `.html` file to include both `jquery.js` and `jasmine-jquery.js` files in the header script section.

Let's have a look at a test that injects DOM elements by using the `jasmine-jquery` library. Firstly, a class that manipulates a specific DOM element:

```
class ModifyDomElement {
  setHtml() {
    var elem = $("#my_div");
    elem.html("<p>Hello world</p>");
  }
}
```

This `ModifyDomElement` class has a single function, named `setHtml` that is using jQuery to find a DOM element with the id of `my_div`. The HTML of this div is then set to a simple "Hello world" paragraph. Now for our Jasmine test:

```
describe("fixture tests", () => {
  it("modifies dom element", () => {
    setFixtures("<div id='my_div'></div>");
    var modifyDom = new ModifyDomElement();
    modifyDom.setHtml();
    var modifiedElement = $("#my_div");
    expect(modifiedElement.length).toBeGreaterThan(0);
    expect(modifiedElement.html()).toContain("Hello");
  });
});
```

The test starts by calling the `jasmine-jquery` function `setFixtures`. This function will inject the HTML provided as the first string argument directly into the DOM. We then create an instance of the `ModifyDomElement` class, and call the `setHtml` function to modify the `my_div` element. We are then setting the variable `modifiedElement` to the result of a jQuery search in the DOM. If jQuery has found the element, then its `length` property will be `> 0`, and we can then check to see if, in fact, the HTML was modified.



The fixture methods provided by `jasmine-jquery` also allow loading raw HTML files off disk, instead of having to write out lengthy string representations of HTML. This is also particularly useful if your MV* framework uses HTML file snippets. The `jasmine-jquery` library also has utilities for loading JSON from disk, and purpose build matchers that work with jQuery. Be sure to check out the documentation at (<https://github.com/velesin/jasmine-jquery>).

DOM events

The `jasmine-jquery` library also adds some Jasmine spies to help with DOM events. If we were creating a button, either within TypeScript code or within HTML, we can ensure that our code correctly responds to DOM events such as `click`. Consider the following code and test:

```
Function handle_my_click_div_clicked() {
    // do nothing at this time
}
describe("click event tests", () => {
    it("spies on click event element", () => {
        setFixtures("<div id='my_click_div' "
            +"onclick='handle_my_click_div_clicked'>Click Here</div>");

        var clickEventSpy = spyOnEvent("#my_click_div", "click");

        $('#my_click_div').click();
        expect(clickEventSpy).toHaveBeenTriggered();
    });
});
```

Firstly, we are defining a dummy function named `handle_my_click_div_clicked`, which is used within the fixture HTML. Having a closer look at the HTML used in the `setFixtures` function call, we are creating a button with an id of `my_click_div`, and an `onclick` DOM event that will call our dummy function. We then create a spy on this click event for the `my_click_div` div, and on the next line actually invoke the click event. Our expectation is using the `jasmine-jquery` matcher `toHaveBeenTriggered` to test whether the `onclick` handler was invoked.



jQuery and DOM manipulation provide us with a way of filling in forms, clicking on **Submit**, **Cancel**, **OK** buttons, and generally simulating user interaction with our application. We can easily write full acceptance or user acceptance tests within Jasmine using these techniques – further solidifying our application against errors and change.

Jasmine runners

There are a number of ways to run Jasmine tests outside of an actual web page, as we have been doing up until this point. Bear in mind, though, that Visual Studio does not support debugging TypeScript outside of directly running a web page with Internet Explorer. In these cases, you would need to revert to the existing developer tools available within your target browser.

Most test runners rely on a simple static HTML page to contain all tests, and will fire up a small instance of a web server in order to serve this HTML page to the test runner. Some test runners use a configuration file for this purpose, and construct a testing environment without the need for HTML at all. This may be all well and good for unit tests – where the integration points of your code are mocked or stubbed – but this approach does not work well for integration or acceptance tests.

Many real-world web applications, for example, run through some server-side business logic to generate HTML for each web request. Authentication logic, for example, may redirect the user to a login page, and then use a forms-based auth cookie on subsequent page requests or RESTful data requests. In these circumstances, running a simple HTML page outside of the actual web application will not work. You need to run your tests within a page that is actually hosted along with the rest of the web application. Also, if you are trying to add a JavaScript test suite to an existing web project, this logic may not be easy to set aside.

For these reasons, we have focused on using a standard HTML page within our web application to run our tests. In an MVC application, for example, we would set up a Jasmine controller, with a `Run` function that returned a `SpecRunner.cshtml` view page. In fact, the default installation of the NuGet package `JasmineTest` will set up these controllers and views as standard templates for us on installation.

Testem

Testem is a Node based command-line utility that will continuously run test suites against connected browsers when it detects that JavaScript files have been modified. Testem is useful for very quick feedback on a number of browsers, and also has a continuous integration flag that can be used on build servers. Testem is suitable for unit testing. More info can be found at the GitHub repository (<https://github.com/airportyh/testem>).

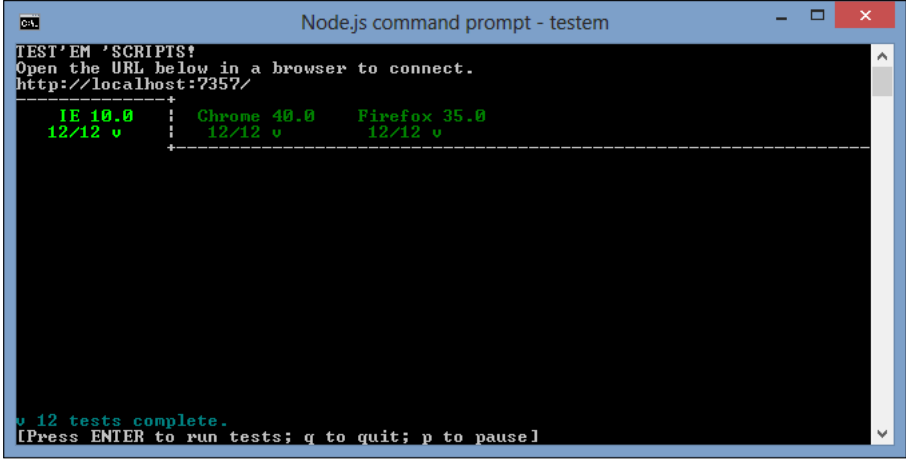
Testem can be installed via Node with the following command:

```
Npm install -g testem
```

To run `testem`, simply navigate to the root folder of your test suite in a command-line window, and type `testem`. Testem will fire up, start a web server, and invite you to connect to it via a browser. In following the screenshots, Testem was running at `http://localhost:7357`. You can connect a number of different browsers to this URL—and Testem will run the specs it finds against each browser. By default, Testem will search the current directory for JavaScript files that contain tests, build an HTML page containing these tests and execute them. If you already have an HTML page that has your tests included, then this page can be specified to Testem via a `testem.yml` config file as follows:

```
{
  "test_page": "tests/01_SpecRunner.html"
}
```

This HTML page will also need to include the `testem.js` file to enable communication with the Testem server.



```
Node.js command prompt - testem
TEST'EM 'SCRIPTS!
Open the URL below in a browser to connect.
http://localhost:7357/
-----+-----+-----+
IE 10.0      Chrome 40.0   Firefox 35.0
12/12 v     12/12 v     12/12 v
-----+-----+-----+
v 12 tests complete.
[Press ENTER to run tests; q to quit; p to pause]
```

Testem output showing three connected browsers

Testem has a number of powerful configuration options that can be specified in the configuration file. Be sure to head over to the GitHub repository for more information.

Note that Testem will not work with ASP.NET MVC controller routes – making it unsuitable for integration testing on ASP.NET MVC sites. If you are using an MVC controller and view to generate your test suite, such that the URL to your running test page is `/Jasmine/Run`, for example – Testem will not work.

Karma

Karma is a test runner built by the Angular team, and features heavily in the Angular tutorials. It is a unit testing framework only, and the Angular team recommends end-to-end or integration tests to be built and run via Protractor. Karma, like Testem, runs its own instance of a web server in order to serve pages and artifacts required by the test suite, and has a large set of configuration options. It can also be used for unit tests that do not target Angular. To install Karma to work with Jasmine 2.0, we will need to install a few packages using `npm`:

```
Npm install karma-jasmine@2_0 -save-dev
```

```
Npm install jasmine-core -save-dev
```

```
Npm install karma-chrome-launcher
```

```
Npm install karma-jasmine-jquery
```

To run Karma, we will firstly need a config file. By convention, this is generally called `karma.conf.js`. A sample karma config file is as follows:

```
module.exports = function (config) {
  config.set({
    basePath: '../..',
    files: [
      'Scripts/underscore.js',
      'Scripts/jquery-1.8.0.js',
      'Scripts/jasmine-jquery/jasmine-jquery.js',
      'Scripts/jasmine-data-provider/SpecHelper.js',
      'tests/*.js'
    ],
    autoWatch: true,
    frameworks: ['jasmine'],
    browsers: ['Chrome'],
    plugins: [
      'karma-chrome-launcher',
```

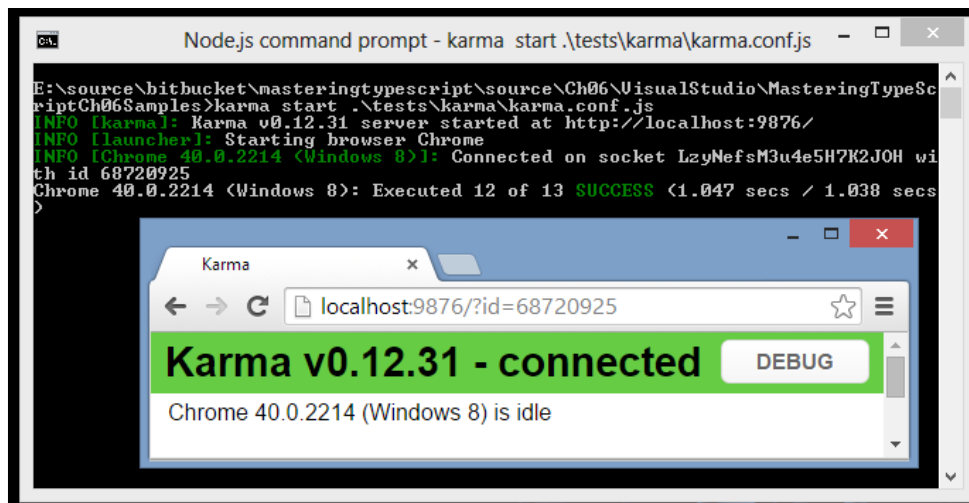
```

    'karma-jasmine'
  ],
  junitReporter: {
    outputFile: 'test_out/unit.xml',
    suite: 'unit'
  }
});
};

```

All config to Karma must be passed in via the `module.exports` and `config.set` convention, as seen in the first two lines. The `basePath` parameter specifies what the root path is of the web project, and is relevant to the directory that the `karma.config.js` file resides in. The `files` array contains list of files to be included in the generated HTML file, and can use the `***.js` matching algorithms to load an entire directory and sub-directory of JavaScript files. The `autoWatch` parameter keeps karma running in the background, watching files for changes, in a similar manner to Testem. Karma also allows for a variety of browsers to be specified – each with their own launcher plugins. Finally, the `junitReporter` is being used in this example to report tests back to a Jenkins CI server. Once this config file is in place, simply run karma start as follows:

```
karma start <path to karma.config.js>.
```



Karma output from a simple test

Protractor

Protractor is a Node based test runner that tackles end-to-end testing. It was originally designed for Angular apps, but can be used with any website. Unlike Testem and Karma, Protractor is able to browse to a specific page and then interact with the page from JavaScript – making it suitable for integration testing. It can check metadata properties such as the page title, or fill in forms and click on buttons, and allow the backend server to redirect to different pages. Protractor documentation can be found here (<https://github.com/angular/protractor>), and can be installed with npm:

```
Npm install -g protractor
```

We will get to running Protractor a little later, but first, let's discuss the engine that Protractor uses in order to automate web pages.

Using Selenium

Selenium is a driver for web browsers. It allows programmatic remote control of web browsers, and can be used to create automated tests in Java, C#, Python, Ruby, PHP, Perl, and even JavaScript. Protractor uses Selenium under the covers to control web browser instances. To install the Selenium server for use with Protractor, run the following command:

```
Webdriver-manager update
```

To start the Selenium server, run the following command:

```
Webdriver-manager start
```

If all goes well, Selenium will report that the server has started, and will detail the address of the Selenium server. Check your output for a line similar to the following:

```
RemoteWebDriver instances should connect to: http://127.0.0.1:4444/wd/hub
```



You will need Java to be installed on your machine in order to run the Selenium server, as the webdriver-manager script uses Java to start the Selenium server.

Once the server is running, we will need a configuration file for Protractor (named `protractor.conf.js`) that includes some settings. At this stage, all we need is the following:

```
exports.config = {
  seleniumAddress: 'http://localhost:4444/wd/hub',
  specs: ['*.js']
}
```

These protractor settings simply set the `seleniumAddress` to the address of the Selenium server, as reported earlier. We also have a `specs` property, which is set to look for any `.js` file within the same directory as the `protractor.conf.js`, and treat them as test specs.

Now for the simplest of tests:

```
describe("simple protractor test", () => {
  it("should navigate to a page and find a title", () => {
    browser.driver.get('http://localhost:64227/Jasmine/Run');
    expect(browser.driver.getTitle()).toContain("Jasmine");
  });
});
```

Our test starts by opening the page at `/Jasmine/Run`. Note that this is an ASP.NET MVC path that uses the default Jasmine controller, and returns `Views/Jasmine/SpecRunner.cshtml`. This controller and view was included with the Jasmine NuGet package that we installed earlier. Make sure that you can navigate to this page in your browser before trying to execute the Protractor tests.

Running Protractor with the configuration file will now execute our previous test:

```
protractor .\tests\protractor\protractor.conf.js
```

And will produce the desired result:

```
Using the selenium server at http://localhost:4444/wd/hub.
Finished in 1.606 seconds
1 test, 1 assertion, 0 failures
```



There are two things that must be running here in order for this test to work:

The Selenium server must be running in a command prompt, such that `localhost : 4444/wd/hub` is a valid address, and does not return 404 errors

The developer ASP.NET website must be up and running so that `localhost : 64277/Jasmine/Run` hits our Visual Studio Jasmine controller, and renders an HTML page

Integration tests

Let's assume that we are conducting integration tests in a test page that is rendered using ASP.NET MVC routes. We want to use the standard MVC controller, action, view method of generating an HTML page, as we may need to execute some server-side logic to setup pre-requisites before the integration tests can start.

Note that in a real-world application, it is often necessary to run server-side logic or use server-side HTML rendering for integration tests. For instance, most applications will require some sort of authentication before allowing calls to REST services via JavaScript. Implementing an `[Authorize]` attribute to your RESTful API controllers is the logical solution. Unfortunately, calling any of these REST controllers from a normal HTML page will return 401 (Unauthorized) errors. One way around this is to use an MVC controller to serve the test HTML page, and then to set up a dummy forms authentication ticket in the server-side code. Once this is in place, any calls to RESTful services from this page will already be authenticated with a dummy user profile. This technique can also be used to run integration tests where users have different roles and different permissions based on their authentication credentials.

Simulating integration tests

To simulate this sort of integration test page, let's reuse the `JasmineController` that was installed with the Jasmine NuGet package. As mentioned earlier, an integration test will need to hit the backend server-side logic (in this case the Jasmine MVC controller), and then render a server-side-generated HTML page to the browser (in this case the `SpecRunner.cshtml` view). This simulation means that we are relying on the server-side MVC framework to resolve the `/Jasmine/Run` URL, generate an HTML page on the fly, and return this generated HTML page to the browser.

This `SpecRunner.cshtml` file (the MVC template for generating the HTML) is very simple:

```
{
  Layout = null;
}
<!DOCTYPE HTML PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN"
  "http://www.w3.org/TR/html4/loose.dtd">
<html>
<head>
  <title>Jasmine Spec Runner</title>

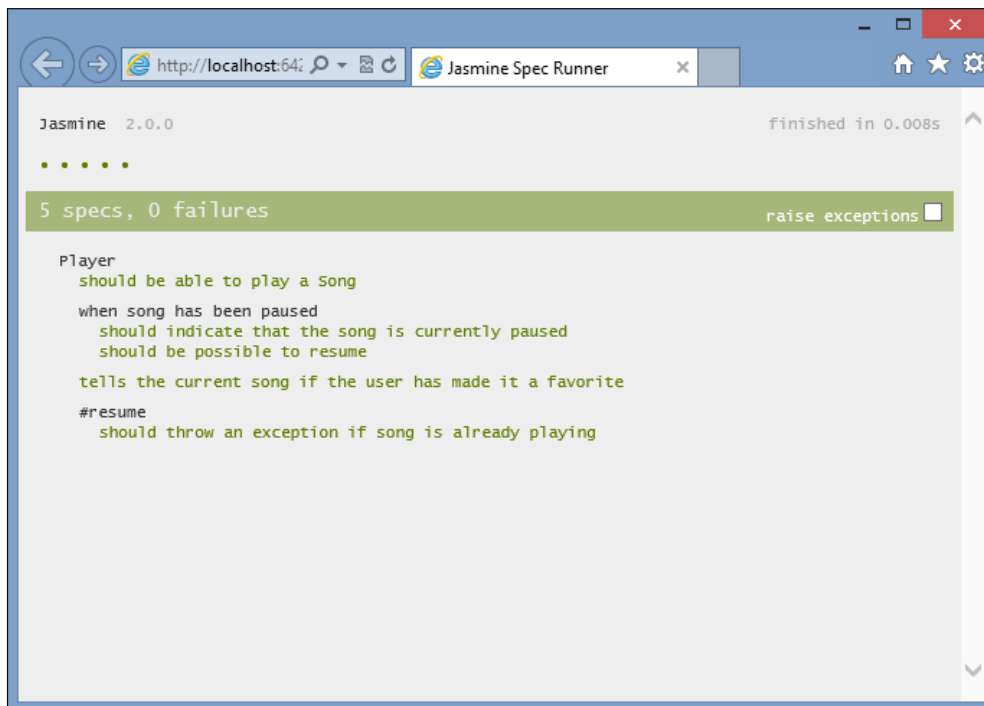
  <link rel="shortcut icon" type="image/png"
    href="/Content/jasmine/jasmine_favicon.png">
  <link rel="stylesheet" type="text/css"
    href="/Content/jasmine/jasmine.css">
  <script type="text/javascript"
    src="/Scripts/jasmine/jasmine.js"></script>
  <script type="text/javascript"
    src="/Scripts/jasmine/jasmine-html.js"></script>
  <script type="text/javascript"
    src="/Scripts/jasmine/boot.js"></script>

  <!--include source files here... -->
  <script type="text/javascript"
    src="/Scripts/jasmine-samples/SpecHelper.js"></script>
  <script type="text/javascript"
    src="/Scripts/jasmine-samples/PlayerSpec.js"></script>

  <!--include spec files here... -->
  <script type="text/javascript"
    src="/Scripts/jasmine-samples/Player.js"></script>
  <script type="text/javascript"
    src="/Scripts/jasmine-samples/Song.js"></script>
</head>

<body>
</body>
</html>
```

This ASP.NET MVC view page is using Razor syntax, and is not based on a master page—as the `Layout` parameter at the top of the file is set to `null`. The page includes a number of links in the head element, including `jasmine.css`, `jasmine.js`, `jasmine-html.js`, and `boot.js`. These are the required Jasmine files that we have seen before. After this, we have just included the `SpecHelper.js`, `PlayerSpec.js`, `Player.js`, and `Song.js` files from the `jasmine-samples` directory. Running this page by navigating to the `/Jasmine/Run` URL will run the sample tests included with Jasmine:



Output of the default/`Jasmine/Run` web page

Our simulated integration test page in this sample just runs a couple of standard Jasmine tests. Using a server-side generated HTML page now allows us to use dummy authentication, if needed. With dummy authentication in place, we can start to write Jasmine tests to target secure RESTful data services.

In our next chapter, we will have a look at building and testing some Backbone models and collections, and will work through further examples of integration tests that actually request data from the server. For the time being, though, we have a sample page that is generated server-side, and that can be used as the base for further integration tests.



Test pages like these should never be packaged in **User Acceptance Testing (UAT)** or release configurations. In ASP.NET, we can simply use a compiler directive such as the `#if DEBUG ... #endif` around our controller classes to exclude them from any other build configuration.

Detailed test results

So we now have the beginnings of an integration test page that shows us the results of our Jasmine test run. This HTML page is good for a quick overview, but we would now like some more detailed information on each test so that we can report back to our build server; how long each test took, and its `success / fail` state.

For these reporting purposes, Jasmine includes the ability to use custom test reporters, over and above the standard `HtmlReporter` that is the Jasmine default. The GitHub project, `jasmine-reporters` (<https://github.com/larrymyers/jasmine-reporters>), has a number of prebuilt test reporters that cater for the most popular build servers. Unfortunately, this project does not have a corresponding NuGet package, so we will need to install the `.js` files within our project manually.



An alternative method of managing JavaScript libraries is the **Bower** package manager. Bower is a Node based command-line utility that is similar to NuGet, but deals only with JavaScript libraries and frameworks.

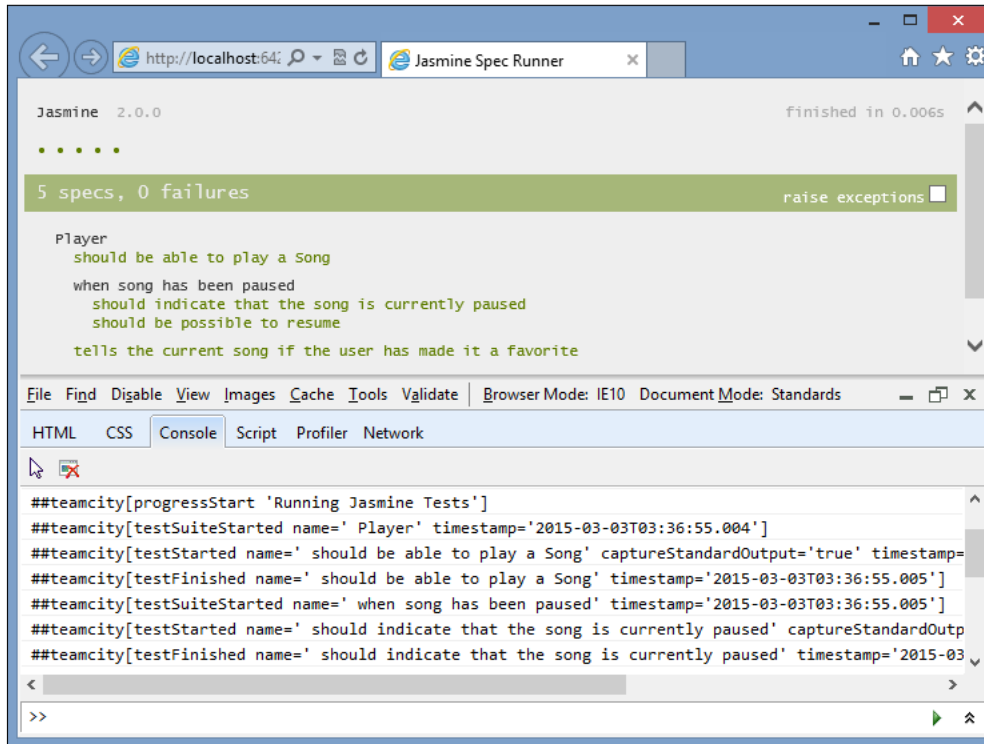
Let's now modify our HTML page to include the TeamCity reporter. Firstly, modify the `SpecRunner.cshtml` file to include a script tag for the `teamcity_reporter.js` file as follows:

```
<script type="text/javascript"
      src="/Scripts/jasmine-reporters/teamcity_reporter.js">
</script>
```

Next, we need to create a simple script within the `body` tag to register this reporter with Jasmine:

```
<script type="application/javascript">
  window.tcapi = new jasmineReporters.TeamCityReporter({});
  jasmine.getEnv().addReporter(window.tcapi);
</script>
```


This script simply creates an instance of the `TeamCityReporter` class, and assigns it to a variable named `tcapi` on the window object. The second line of this script adds this reporter to the Jasmine environment. Running our page now will produce TeamCity results logged to the console:



Jasmine output with TeamCity messages logged to the console

Logging test results

We now need to access this output, and find a way to report it back to the Protractor instance. Unfortunately, accessing the console's log through Selenium will only report critical errors, so the preceding TeamCity reporter output will be unavailable. A quick look around the `teamcity_reporter.js` code reveals that all `console.log` output messages use the `tclog` function to build a string, and then call `console.log` with this string. As we have an instance of our `TeamCityReporter` available to us, we can easily store these logged messages into an array, and then read through them once the test suite has finished running. Some quick modifications to the JavaScript file `teamcity_reporter.js` are as follows.

Just under the constructor function for the `TeamCityReporter` class, create an array:

```
exportObject.TeamCityReporter = function (args) {
    self.logItems = new Array();
}
```

Now we can modify the `tclog` function to return the string that it is building:

```
Function tclog(message, attrs) {
    log(str); // call to console.log
    return str; // return the string to the calling function
}
```

Then, each call to `tclog` can push the returned string to this array:

```
self.jasmineStarted = function (summary) {
    self.logItems.push(
        tclog("progressStart 'Running Jasmine Tests'"));
};
```

Now that the `TeamCityReporter` has a `logItems` array, we will need some method of finding out when the test suite has finished, and we can then loop through the array of `logItems`, and attach them to the DOM. Once it is in the DOM, our Protractor instance can use Selenium to read these values and report back to the command line.

Let's build a small class named `JasmineApiListener` that accepts an instance of the `TeamCityReporter` class to do all this work for us:

```
class JasmineApiListener {
    private _outputComplete: boolean;
    private _tcReporter: jasmine.ITeamCityReporter;

    constructor(tcReporter: jasmine.ITeamCityReporter) {
        this._outputComplete = false;

        this._tcReporter = tcReporter;
        var self = this;

        window.setInterval(() => {
```

```
    if (self._tcReporter.finished &&
        !self._outputComplete) {
        var logItems = self._tcReporter.logItems;
        var resultNode = document.getElementById(
            'teamCityReporterLog');
        resultNode.setAttribute('class',
            'teamCityReporterLog');
        for (var I = 0; I < logItems.length; i++) {
            var resultItemNode =
                document.createElement('div');
            resultItemNode.setAttribute('class', 'logentry');
            var textNode =
                document.createTextNode(logItems[i]);
            resultItemNode.appendChild(textNode);
            resultNode.appendChild(resultItemNode);
        }
        self._outputComplete = true;

        var doneFlag = document.getElementById(
            'teamCityResultsDone');
        var doneText = document.createTextNode("done");
        doneFlag.appendChild(doneText);
    }

    }, 3000);
}

}
```

Our `JasmineApiListener` class has two private variables. The `_outputComplete` variable is a boolean flag indicating that the test suite has completed, and that the results have been written to the DOM. The `_tcReporter` variable holds an instance of the `TeamCityReporter` class, which is passed through in the constructor. The constructor simply sets the flag `_outputComplete` to false, creates a variable named `self`, and sets up a simple timer on a three-second interval.



The `self` variable is a necessary scoping step in order to access the correct instance of `this` inside the anonymous function that is passed to `setInterval`.

The body of our anonymous function is where all the goodness takes place. Firstly, we are checking the `_tcReporter.finished` property on the `TeamCityReporter` instance to tell whether or not the suite has completed. If it has, and we have not yet appended our results to the DOM (`!self._outputComplete`), then we can access the `logItems` array and create DOM elements for each of these entries. These elements are attached as `<div class="logentry">...</div>` elements, as children of the parent `<div id="teamCityReporterLog">` element.

Note that the preceding code is using the native `document.getElementById` and `appendChild` syntax for DOM manipulation, and not a jQuery-style syntax to avoid having a dependency on jQuery.

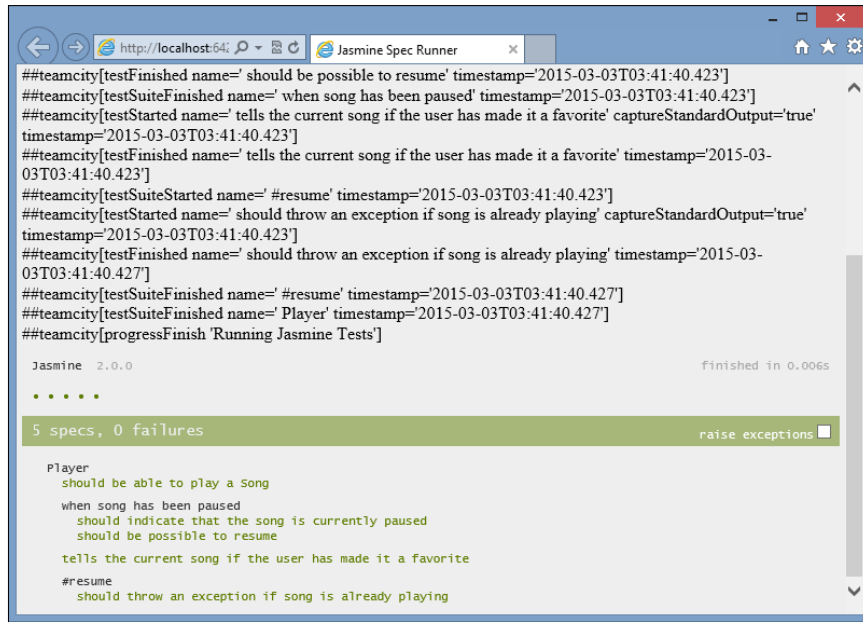
We can now modify the script within the `SpecRunner.cshtml` view as follows:

```
<script type="application/javascript">
    window.tcapi = new jasmineReporters.TeamCityReporter({});
    jasmine.getEnv().addReporter(window.tcapi);
    var jasmineApiListener = new JasmineApiListener(window.tcapi);
</script>

<div id="teamCityResultsDone"></div>
<div id="teamCityReporterLog"></div>
```

The first script is the updated version of what we have been using previously, which now creates an instance of our `JasmineApiListener` class, and passes the instance of the `TeamCityReporter` class within the constructor. We have also added two `<div>` tags. The first one, `teamCityResultsDone`, is a flag to indicate that we have completed writing the TeamCity results to the DOM, and the second `teamCityReporterLog` is the parent `div` to hold all of the child `logentry` elements.

If we fire up this page now, we should see our tests run through, and then three seconds later, the DOM will be updated with the results that we have read from the `TeamCityReporter` array, as shown in the following screenshot:



Jasmine output being logged to the DOM

Now that we have a way of logging the results of our tests to the DOM, we can update our Protractor based Selenium tests to relate these results to our build server.

Finding page elements

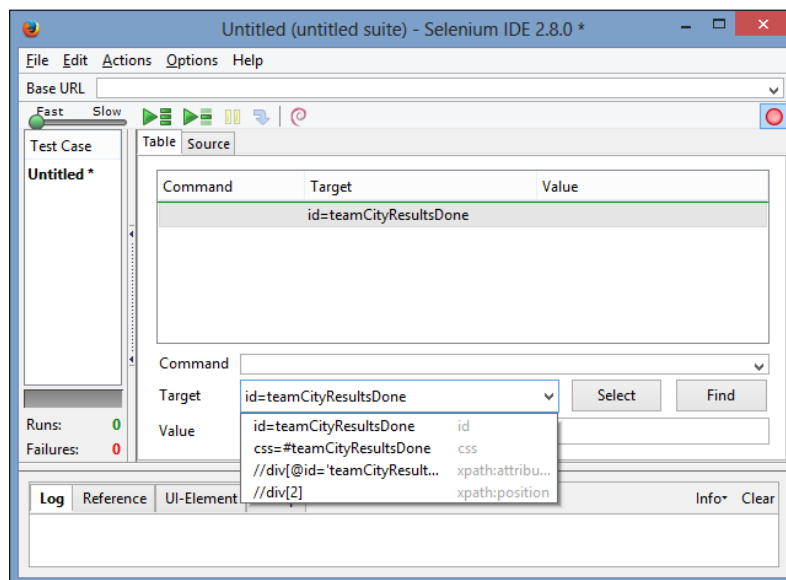
As mentioned previously, Protractor can be used to run integration tests, as well as automated acceptance tests. A Protractor test can browse to a login page, find the login username textbox, send a value such as "testuser1" to this textbox, and then repeat the process for a password. The same test code can then be used to click on the **Login** button, which will submit the form to our server login controller. Our test can then ensure that the server responds with the correct redirect to our main page. This main page may contain multiple buttons, grids, images, side panels and navigation elements. Ideally, we would want to write acceptance tests for each of these page elements.

Protractor uses locators to find these elements within our DOM. These elements can be found by their CSS selectors, by `id`, or, if using Angular, by model or binding. Building the correct strings for use in these selectors can sometimes be difficult.

Selenium provides us with a useful Firefox extension to help when writing Selenium based tests - the Selenium IDE (<http://docs.seleniumhq.org/projects/ide/>). With this extension installed within Firefox, we can use the IDE to help find elements on the page.

As an example of how to use this extension, let's continue with our work on the Jasmine reporter that we have writing, and find the `teamCityResultsDone` DOM element that we have been using to flag a completed test suite. The code and process we use to find this DOM element is the same code and process that we would use to find other page elements on a login page, for example, or any other page that we were driving through Selenium.

If we fire up our `/Jasmine/Run` page using Firefox, we can now click on the Selenium IDE button on the top right of the browser to launch the Selenium IDE. This IDE uses commands to record interactions against a web page, and shows this list of commands in the main window. Right-click on the command window, and select **Insert new command**. In the command name text box give the new command a name—something like `find done element`. Once a command has a name, the two buttons next to the target input box become enabled, and we can click on **Select**. We can then drag our mouse over the web page, and click on the text **done** at the top of the page. Notice how the command has automatically filled in the **Target** element in the Selenium IDE. The **Target** input box has now become a drop-down list, and we can use this list to show the Selenium selector syntax for our `teamCityResultsDone` `div`, as shown in the following screenshot:



Firefox Selenium IDE

Working with page elements in Jasmine

Now that we know how to find an HTML page element using the Selenium IDE, we can start writing Selenium commands to query the page elements of our Jasmine tests. Remember that there are two elements we need to find.

Firstly, we need to find the `teamCityResultsDone` div, and wait for the text of this element to be updated. This div is only updated when our Jasmine test suite is complete, and our tests results have been included in the DOM. Once our test suite has been flagged as complete, we then need to loop through each of the `logentry` divs that are child elements of the `teamCityReporterLog` div. These `logentry` divs will contain the detailed results each of our tests.

The changes needed in our protractor tests are as follows:

```
describe("team city reporter suite", () => {
  it("should find test results", () => {
    browser.driver.get('http://localhost:64227/Jasmine/Run');

    expect(browser.driver.getTitle()).toContain("Jasmine");

    var element = browser.driver.findElement(
      { id: "teamCityResultsDone" });

    browser.driver.wait(() => {
      return element.getText().then((value) => {
        return value.length > 0;
      });
    }, 60000, "failed to complete in 60 s");
  });

  afterEach(() => {
    browser.driver.findElements(
      by.css("#teamCityReporterLog > div.logentry")
    ).then((elements) => {
      for (var i = 0; i < elements.length; i++) {
        elements[i].getText().then((textValue) => {
          console.log(textValue);
        });
      }
    });
  });
});
```

Our test begins by browsing to the `/Jasmine/Run` page, and expects this page title to contain "Jasmine", as we have seen previously. We are then using the `findElement` function from Selenium to find an element on the page. This function is passed a JavaScript object with the `id` set to `teamCityResultsDone`—and is using the `select` syntax that we saw earlier in the Selenium IDE.

We are then calling the `wait` function to wait for the text of the `teamCityResultsDone` element to be updated (that is, its `length` is `> 0`), and set a 60-second timeout for this `wait` function. Remember that our `JasmineApiListener` code will set the text value of this div to "done" when we have finished updating the DOM, which will effectively then trigger the `wait` function.

We are then using the `afterEach` function to loop through the `logentry` divs. Instead of finding the parent element, we are now using the `findElements` Selenium function to find multiple elements on the page.

Note the Selenium selector syntax that we are using for these divs:

```
by.css("#teamCityReporterLog > div.logentry").
```

This `by.css` function is using CSS selector syntax to find our elements, and the input string corresponds to the CSS selector that the Selenium IDE shows. We can therefore use the Selenium IDE to help us find the correct CSS selector syntax.

Selenium uses a fluent syntax for most of its API functions. The call to the `findElements`, therefore, is followed by a `.then` function, which will pass the elements it has found in an array to the anonymous function. We use this anonymous function with the `.then((elements) => { .. })` syntax. Within this function, we are looping through each element of the elements array, and calling the `.getText` Selenium function. Again, this `getText` function provides a fluent syntax, which allows us to write another anonymous function to use the text value returned, as seen in the line `elements[i].getText().then((textValue) => { ... });`. This function is simply logging the `textValue` to the protractor console.

Running our Protractor test will now report our test results to the command line as follows:

```
λ protractor .\tests\protractor\protractor.conf.js
Using the selenium server at http://localhost:4444/wd/hub
##teamcity[progressStart 'Running Jasmine Tests']
##teamcity[testSuiteStarted name=' Player' timestamp='2014-12-06T08:54:27.827']
##teamcity[testStarted name=' should be able to play a Song' captureStandardOutput='true' timestamp='2014-12-06T08:54:27.829']
##teamcity[testFinished name=' should be able to play a Song' timestamp='2014-12-06T08:54:27.832']
##teamcity[testSuiteStarted name=' when song has been paused' timestamp='2014-12-06T08:54:27.832']
##teamcity[testStarted name=' should indicate that the song is currently paused' captureStandardOutput='true' timestamp='2014-12-06T08:54:27.833']
##teamcity[testFinished name=' should indicate that the song is currently paused' timestamp='2014-12-06T08:54:27.834']
##teamcity[testStarted name=' should be possible to resume' captureStandardOutput='true' timestamp='2014-12-06T08:54:27.834']
##teamcity[testFinished name=' should be possible to resume' timestamp='2014-12-06T08:54:27.835']
##teamcity[testSuiteFinished name=' when song has been paused' timestamp='2014-12-06T08:54:27.835']
##teamcity[testStarted name=' tells the current song if the user has made it a favorite' captureStandardOutput='true' timestamp='2014-12-06T08:54:27.836']
##teamcity[testFinished name=' tells the current song if the user has made it a favorite' timestamp='2014-12-06T08:54:27.838']
##teamcity[testSuiteStarted name=' #resume' timestamp='2014-12-06T08:54:27.839']
##teamcity[testStarted name=' should throw an exception if song is already playing' captureStandardOutput='true' timestamp='2014-12-06T08:54:27.840']
##teamcity[testFinished name=' should throw an exception if song is already playing' timestamp='2014-12-06T08:54:27.842']
##teamcity[testSuiteFinished name=' #resume' timestamp='2014-12-06T08:54:27.843']
##teamcity[testSuiteFinished name=' Player' timestamp='2014-12-06T08:54:27.844']
##teamcity[progressFinish 'Running Jasmine Tests']
.
Finished in 4.406 seconds
1 test, 1 assertion, 0 failures
```

Protractor logging test results to the console

Mission accomplished. We are now using Protractor to browse to a server-generated HTML page that runs a set of Jasmine tests. We are then using Selenium to find elements on the page, waiting for DOM updates, and then loop through an array of elements in order to log our Jasmine test results to the protractor console.

These Selenium functions, such as `browser.driver.get`, `findElements`, and `wait` are all part of the rich set of functionality that Selenium provides to work with DOM elements. Be sure to head over to the Selenium documentation for more information.

We now have a mechanism to fire up an integration test page, run a Jasmine test suite, report these test results to the DOM, and then read these results and log them to the Protractor console. It is then a simple matter to set up a build step within a TeamCity build server to execute protractor, and record these test results during the build process.

Summary

In this chapter, we have explored Test Driven Development from the ground up. We have discussed the theory of TDD, explored the differences between unit, integration, and acceptance tests, and had a look at what a CI build server process would look like. We then explored Jasmine as a testing framework, learned how to write tests, use expectations and matchers, and also explored Jasmine extensions to help with data-driven tests and DOM manipulation through fixtures. Finally, we had a look at test runners, and built a Protractor based test framework to drive web pages through Selenium, and report the results back to a build server. In the next chapter, we will explore the TypeScript module syntax, in order to use both CommonJS and AMD JavaScript modules.

7

Modularization

Modularization is a popular technique used in modern programming languages that allows programs to be built from a series of smaller programs, or modules. Writing programs that use modules encourages programmers to write code that conforms to the design principle called "Separation of Concerns". In other words, each module focuses on doing one thing, and has a clearly defined interface. If we then consume this module by focusing on the interface, we can easily replace this interface with something else, without breaking our code. We will focus more on "Separation of Concerns" and other object-oriented design patterns in the next chapter.

JavaScript, in itself, does not have a concept of modules, but it is proposed for the upcoming ECMAScript 6 standard. Popular frameworks and libraries such as Node and Require have built module-loading capabilities into their frameworks. These frameworks, however, use slightly different syntax. Node uses the CommonJS syntax for module loading, whereas Require uses the **Asynchronous Module Loading (AMD)** syntax. The TypeScript compiler has an option to turn on module compilation, and then switch between these two syntax styles.

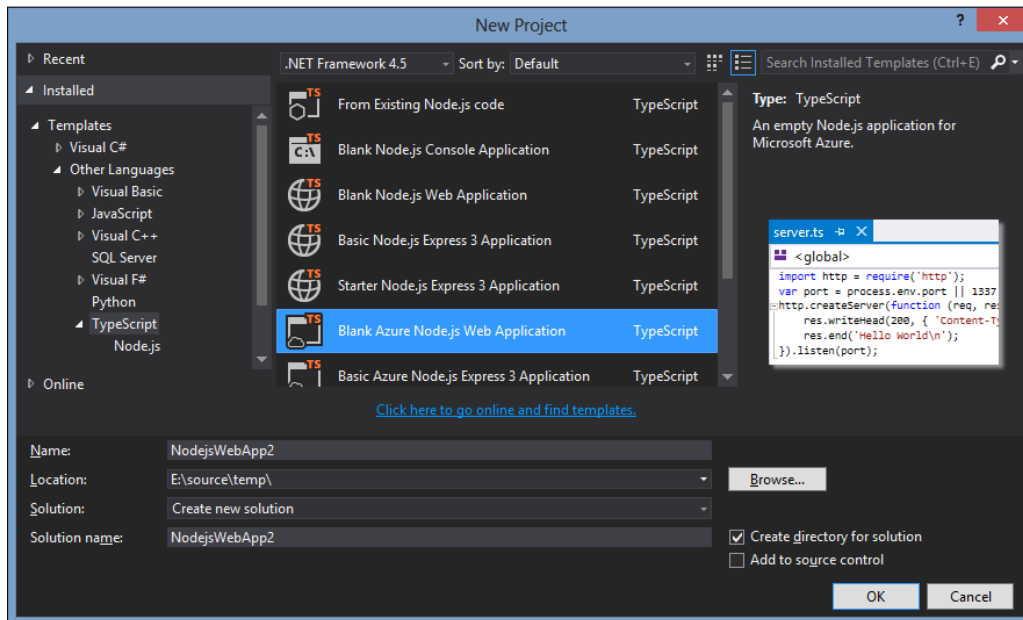
In this chapter, we will look at the syntax of both module styles, and how the TypeScript compiler implements them. We will take a look at how to use modules when writing code for both Node and Require. We will also have a cursory look at Backbone, and how to write an application using a Model, View and Controller. Each of these Backbone components will be built as loadable modules.

CommonJs

The most prevalent usage of the CommonJs syntax for writing modules is when writing server-side code. It has been argued that browser-based CommonJs syntax simply cannot be done without a lot of overhead, but there are some libraries out there such as Curl (<https://github.com/cujojs/curl>) that allow this syntax. In this section, we will, however, focus on Node application development.

Setting up Node in Visual Studio

Using Node within Visual Studio has been made a breeze by the Node tools for Visual Studio plugin (<https://nodejstools.codeplex.com>). This toolset has also been updated to use TypeScript as a default editor, bringing the full TypeScript development experience to Node. Once the extension has been installed, we can create a new blank Node application, as shown in the following screenshot:



Creating a blank Node application with the Node toolset

This project template will create a `server.ts` TypeScript file, and include the `node.d.ts` declaration file automatically for us. If we compile and run this default implementation by simply hitting `F5`, the project template will automatically start up a new console to run our Node server, start the server instance, and open a browser to connect to this instance. If all goes well at this stage, your browser will simply say **Hello World**.

Let's take a look at the `server.ts` TypeScript file that is creating an instance of our Node server:

```
import _http = require('http');
var port = process.env.port || 1337
http.createServer(function (req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}).listen(port);
```

The first line of this code snippet uses the CommonJS module syntax to tell our Node server that it must `import` the library named `'http'`.

This line has two key parts. To explain these key parts, let's start at the right-hand side of the `=` sign and work our way towards the left. The `require` function takes a single parameter, and is used to tell the application that there is a library named `'http'` out there. The `require` function also tells the application that it needs this library to be made available to it, in order to continue functioning. As `require` is a key part of the syntax of modules in TypeScript, it has been given the keyword status and will be highlighted in blue, just like other keywords such as `var`, `string`, and `function`. If the application cannot find this `'http'` library, then Node will immediately throw an exception.

The left-hand side of the `=` sign uses the `import` keyword, which is also a fundamental concept in module syntax. The `import` statement tells the application to attach the library that has been loaded via the `require` function, `require('http')`, into a namespace called `_http`. Whatever functions or objects that the `'http'` library has made public will be available to the program via the `_http` namespace.

If we jump to the third line very quickly, we can see that we invoke a function called `createServer` that is defined in the `'http'` module, and call it via the `_http` namespace. hence `_http.createServer()`.



The default `server.ts` file that is generated by the blank Node project template is very slightly different than our preceding code sample. It names the import `http`, which matches the library name `'http'`, as follows:

```
import http = require('http');
```

This is a common naming standard for Node. You can, of course, name your import namespaces whatever you like, but it does help to have the namespace match the imported library's name, to help with the readability of the code.

The second line of our code snippet simply sets up the variable named `port` to either be the value of the global variable `process.env.port`, or a default value of `1337`. This port number is used on the very last line, and uses fluent syntax to call the `listen` function on the returned value of the `http.createServer` function.

Our `createServer` function has two variables named `req` and `res`. If we use our mouse to hover over the `req` variable, we can see that it is of type `_http.ServerRequest`. Similarly, the `res` variable is of type `_http.ServerResponse`. These two variables are our HTTP request and response streams. In the body of the code, we are invoking the `writeHead` function on the HTTP response to set the content-type, and then we are invoking the `end` function on the HTTP response to write the text `'Hello World\n'` to the browser.

With these couple of lines of code, we have created a running node HTTP server that serves up a simple web page with the text **"Hello World"**.

Note that if you have a keen eye for TypeScript syntax, you will notice that this file uses JavaScript syntax and not TypeScript syntax for our `createServer` function. This is most probably due to the recent upgrade of the Node toolset from JavaScript to TypeScript. The call to `createServer` can also be written using TypeScript fat arrow syntax as follows:

```
_http.createServer((req, res) => { .. })
```

Creating a Node module

To create a Node module, we simply need to create another TypeScript file to house our module code. Let's create a file named `ServerMain.ts`, and move the code that writes to the HTTP response into this module as follows:

```
import http = require('http');
export function processRequest(
  req: http.ServerRequest,
  res: http.ServerResponse): void
{
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}
```

Our `ServerMain` module starts with the import of the `'http'` module into the `http` namespace. This is necessary to allow us to use the `ServerRequest` and `ServerResponse` types that are part of this library.

The keyword `export` is now used to indicate what functions will be made available to users of this module. As we can see, we have exported a function named `processRequest` that takes two parameters, `req` and `res`. This function will be used as a replacement for the anonymous function `(req, res) => { ... }` that we were using in the `server.ts` file previously.

Note that as good TypeScript coders, we have also strongly typed the `req` and `res` variables to be of type `http.ServerRequest`, and of type `http.ServerResponse` respectively. This will enable Intellisense within our IDE, and also adheres to two principles of strong typing (S.F.I.A.T and self-describing functions).

Before we modify the `server.ts` file to use our new module, let's crack open the generated JavaScript file and take a closer look at the CommonJs syntax in a little more detail:

```
function processRequest(req, res) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  res.end('Hello World\n');
}
exports.processRequest = processRequest;
```

The first part of this JavaScript is simple enough – we have a function named `processRequest`. The last line, however, attaches this function to a property on the `exports` global variable. This `exports` global variable is how CommonJs publishes modules to the outside world. Any function, class, or property that needs to be exposed to the outside world must be attached to the `exports` global variable. The TypeScript compiler will generate this line of code for us whenever we use the `exports` keyword in a TypeScript file.

Using a Node module

Now that we have our module in place, we can modify our `server.ts` file to use this module as follows:

```
import http = require('http');
import ServerMain = require('./ServerMain');
var port = process.env.port || 1337;
http.createServer(ServerMain.processRequest).listen(port);
```

The first line stays the same, but the second line uses the same `import` and `require` syntax to now import our `./ServerMain` module into the `ServerMain` namespace.



The syntax that we use to name this module points to a local file module, and therefore uses a relative file path to the module file. This relative path will resolve to the `ServerMain.js` file that TypeScript has generated. Creating a global Node module with the name `'ServerMain'`, which would be globally available—similar to the `'http'` module—is outside the scope of this discussion.

Our call to the `http.createServer` function now passes in our `processRequest` function as an argument. We have changed from an anonymous function using the fat arrow syntax, to a named function from the `ServerMain` module. We have also started to adhere to our "Separation of Concerns" design pattern. The `server.ts` file starts the server on a specific port, and the `ServerMain.ts` file now houses the code used to process a single request.

Chaining asynchronous functions

When writing Node code, it is necessary to take a careful note of the asynchronous nature of all Node programming, as well as JavaScript's lexical scoping rules. Luckily, the TypeScript compiler will generate errors if we break any of these rules. As an example of this, let's update our `ServerMain` module to read in a file from disk, and serve up the contents of this file, instead of our `Hello world` text, as follows:

```
import fs = require("fs");
export function processRequestReadFromFileAnonymous(
  req: http.ServerRequest, res: http.ServerResponse)
{
  fs.readFile('server.js', 'utf8', (err, data) => {
    res.writeHead(200, { 'Content-Type': 'text/plain' });
    if (err)
      res.write("could not open file for reading");
    else {
      res.write(data);
      res.end();
    }
  });
}
```

To read files from disk, we will need to use the Node global module named "fs", or filesystem, which is imported on the first line of the code. We then expose a new function named `processRequestReadFromFileAnonymous` that again uses the `req` and `res` parameters. Within this function, we then use the `fs.readFile` function to read a file from disk using three arguments. The first argument is the name of the file to be read in, the second argument is the file type, and the third argument is a callback function that Node will call, once the file has been read from disk.

The body of this anonymous function is similar to what we have seen previously, but it also checks the `err` argument to see whether there was an error while loading the file. If there was no error, the function simply writes the file to the response stream.

In real-world applications, the logic inside of the main `processRequestReadFromFileAnonymous` function could become quite complex (besides the name), and may involve more than a single step to read a hardcoded filename from disk. Let's move this anonymous function into a private function, and see what happens. Our first pass at refactoring this code may be something similar to the following:

```
export function processRequestReadFromFileError(
  req: http.ServerRequest, res: http.ServerResponse)
{
  fs.readFile('server.js', 'utf8', writeFileToStreamError);
}
function writeFileToStreamError(err, data) {
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  if (err)
    res.write("could not open file for reading");
  else {
    res.write(data);
    res.end();
  }
}
```

Here, we have modified the `fs.readFile` function call, and replaced the anonymous callback function with a named function—`writeFileToStreamError`. This change, however, will immediately generate a compilation error:

```
Cannot find name 'res'.
```

This compilation error is caused by the lexical scoping rules of JavaScript. The function `writeFileToStreamError` is trying to use the `res` parameter of the parent function. However, as soon as we moved this function outside the lexical scope of the parent, the variable `res` is no longer in scope – and will therefore be undefined. To fix this error, we need to ensure that the lexical scope of the `res` argument is maintained within our code structure, and we need to pass the value of the `res` argument down to our `writeFileToStream` function, as follows:

```
export function processRequestReadFromFileChained(
  req: http.ServerRequest, res: http.ServerResponse)
{
  fs.readFile('server.js', 'utf8', (err, data) => {
    writeFileToStream(err, data, res);
  });
}
function writeFileToStream(
  err: ErrnoException, data: any,
  res: http.ServerResponse): void
{
  res.writeHead(200, { 'Content-Type': 'text/plain' });
  if (err)
    res.write("could not open file for reading");
  else {
    res.write(data);
    res.end();
  }
}
```

Note that in the call to `fs.readFile` on the third line in the preceding code, we have reverted back to our anonymous syntax, and passed on the value of the parent `res` argument down to our new function `writeFileToStream`. This modification of our code now correctly adheres to the lexical scoping rules of JavaScript. Another side-effect is that we have clearly defined what variables the `writeFileToStream` function needs, in order to work. It needs the `err` and `data` variables from the `fs.readFile` callback, but it also needs the `res` variable from the original HTTP request.

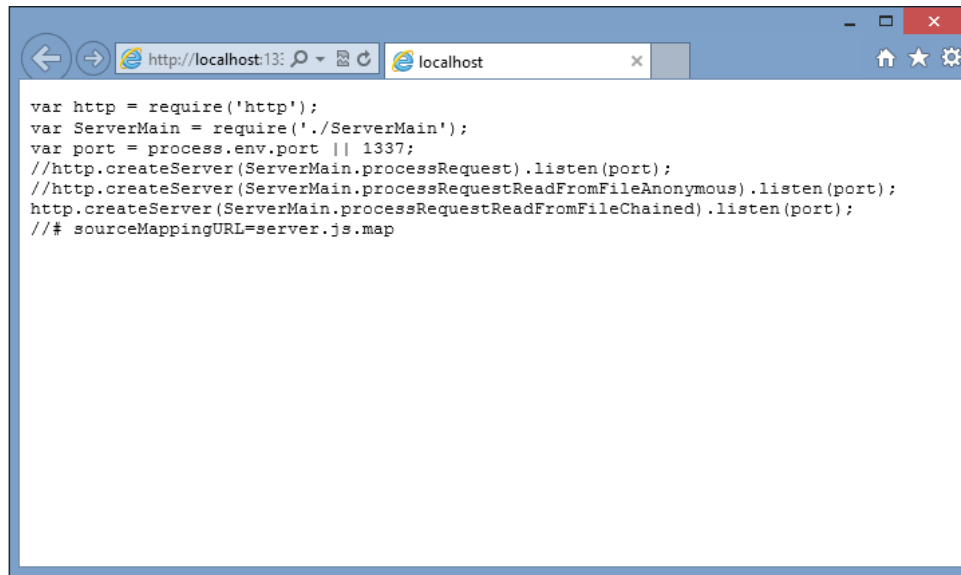


We have not exported the `writeFileToStream` function; it is purely an internal function for use within our module.

We can now modify our `server.ts` file to use our new chained function:

```
http.createServer(ServerMain.processRequestReadFromFileChained)
  .listen(port);
```

Running the application now will show the world what our `server.js` file contains:

A screenshot of a web browser window. The address bar shows 'http://localhost:1337'. The page content displays the following JavaScript code:

```
var http = require('http');
var ServerMain = require('./ServerMain');
var port = process.env.port || 1337;
//http.createServer(ServerMain.processRequest).listen(port);
//http.createServer(ServerMain.processRequestReadFromFileAnonymous).listen(port);
http.createServer(ServerMain.processRequestReadFromFileChained).listen(port);
///

```
sourceMappingURL=server.js.map
```


```

The Node application serving the contents of a file on disk

Note that because we are using modules, we have been able to write three different versions of the `processRequest` function, each with a slight twist. However, our modifications to the `server.ts` file that launches the server have been very simple. We have just replaced the function that the server calls, in order to effectively run three different versions of our application. Again, this complies with the "Separation of Concerns" design principle. The `server.ts` code is simply used to start the Node server on a specific port, and should not be concerned with how each request is processed. Our code within `ServerMain.ts` is responsible simply for processing a request.

This concludes our section on writing Node applications within TypeScript. As we have seen, the TypeScript developer experience brings with it a compilation step, which will quickly trap lexical scoping rules and many other issues within our code. Final score, TypeScript: 1, buggy code: 0!

Using AMD

AMD stands for Asynchronous Module Definition, and as the name suggests, loads modules asynchronously. This means that when an HTML page is loaded, requests to fetch the JavaScript module files happen at the same time. This allows our page to load faster, as we are requesting smaller amounts of JavaScript simultaneously.

AMD module loading is typically used in browser applications, and works together with third-party libraries that provide a script-loading capability. One of the most popular script and module loaders currently available is Require. In this section, we will look at how to use the AMD module-loading syntax, and how to implement Require in a browser-based application.

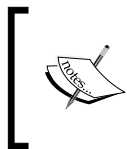
To begin with, let's create a simple TypeScript-based solution, using the "**Html application with TypeScript**" Visual Studio template. If you are not using Visual Studio, then simply create a new project or base source directory, and set up your environment for TypeScript compilation. To use AMD compilation, we will need to set our TypeScript project properties in order to compile to the AMD module syntax.

Using NuGet, we will then install the following packages:

- `RequireJS`
- `Requirejs.TypeScript.DefinitelyTyped`
- `jQuery`
- `jquery.TypeScript.DefinitelyTyped`
- `JasmineTest`
- `Jasmine.TypeScript.DefinitelyTyped`

We will also base our application on Backbone, so we will therefore need the following NuGet packages:

- `Backbone.js`
- `Backbone.TypeScript.DefinitelyTyped`



The Backbone installation will also install Underscore, and the `Backbone.TypeScript.DefinitelyTyped` package will also install `underscore.TypeScript.DefinitelyTyped`.

Backbone

Backbone provides a very minimalistic framework for writing rich client-side JavaScript applications. It uses the MVC pattern to abstract our logic away from direct DOM manipulation. Backbone provides a core set of functionality that is broken up into models, collections, and views, as well as some utility classes to help with events and routing. The library itself is incredibly small, with the minimized `.js` file under 20 KB in size. Its only dependency is Underscore, which is a utility library, again under 16 KB in size. Backbone is a very popular library, has a huge number of extensions, and is relatively easy to learn and implement.

Models, collections and views

At the core of Backbone lies the model. A model is a class that has a set of properties, and represents an item of information that will be treated as a unit. You could think of a model as a single row of data in a database table, or as an object to hold a particular type of information. Model objects are typically very simple, with a few getters and setters for each of their properties, and possibly a `url` property for use with RESTful services. Arrays of models are held within a collection. A collection could be thought of as all the rows of data in a database table, or a logical group of models, each of the same type. Models can contain other models, and can also contain collections, so we are free to mix and match and combine collections and models at will.

Models, therefore, are used to define the structure of the data that our application uses. Backbone provides a simple `url` property for both models and collections, which is used to synchronize Backbone models with RESTful services. Backbone will take care of generating create, read, update, and delete AJAX calls to our services via this `url` property.

Once a model or collection has been created, it is then passed to a view. A Backbone view is responsible for combining the properties of a model with an HTML template. Templates are made up of normal HTML, but have a special syntax to allow the properties of a model to be injected into this HTML. Once this HTML template has been combined with a model, the view can render the resultant HTML to the page.

Backbone does not really have the concept of a controller, as found in the classic MVC definition, but we can use normal TypeScript classes to accomplish the same functionality.

Creating a model

Let's dive right into Backbone, and start with the definition of a model. In this sample, we will work with the concept of a contact – that simply has a `Name` and `EmailAddress` property – as shown in the following code.

Note that this `ContactModel.ts` file is located under the `/tscode/app/models` directory:

```
interface IContactModel {
    Name: string;
    EmailAddress: string;
}
export class ContactModel extends Backbone.Model
    implements IContactModel
{
    get Name() {
        return this.get('Name');
    }
    set Name(val: string) {
        this.set('Name', val);
    }
    get EmailAddress() {
        return this.get('EmailAddress');
    }
    set EmailAddress(val: string) {
        this.set('EmailAddress', val);
    }
}
```

We start with the definition of an interface named `IContactModel`, which has our `Name` and `EmailAddress` properties, both of which are strings.

Next, we create a class named `ContactModel` that derives from, or *extends*, the base `Backbone.Model` class. Note that we are using the `export` keyword before our class definition, to indicate to the TypeScript compiler that we are creating a module that can be imported elsewhere. The `export` keyword and usage is exactly the same as what we have seen previously when we used the CommonJS syntax. Our `ContactModel` class implements the `IContactModel` interface, and also uses ES5 `get` and `set` syntax to define the `Name` and `EmailAddress` properties.



The implementation of each of these properties calls the Backbone `this.get('<propertyname>')` or `this.set('<propertyname>', value)` functions. Backbone stores model properties as object attributes, and uses these `get` and `set` functions internally to interact with model properties – hence the syntax used previously.

Let's follow TDD practices, and write a set of unit tests to make sure that we can create an instance of our `ContactModel` correctly. For this test, we will create a `ContactModelTests.ts` file under the `/tscode/tests/models` directory, as follows:

```
import cm = require("../..../app/models/ContactModel");
describe('/tests/models/ContactModelTests', () => {
  var contactModel: cm.ContactModel;
  beforeEach(() => {
    contactModel = new cm.ContactModel(
      { Name: 'testName',
        EmailAddress: 'testEmailAddress'
      });
  });
  it('should set the Name property', () => {
    expect(contactModel.Name).toBe('testName');
  });
  it('should set the Name attribute', () => {
    expect(contactModel.get('Name')).toBe('testName');
  });
});
```

The first line of this test uses the `import <namespace> = require('<filename>')` syntax that we have seen previously, to import the `ContactModel` module that we exported earlier. You will notice that the file name uses a relative path, which drops down two directories (`"../..../"`) before specifying the `"app/models/ContactModel"` path. This is because AMD module compilation uses paths that are relative to the current file. As our test code is in the `/tscode/tests/models` directory, this relative path must point to the correct directory that contains the `ContactModel.ts` TypeScript file.

Our test defines a variable named `contactModel` that is strongly typed to be of type `cm.ContactModel`. Again, we are using the prefix from the `import` statement as a namespace in order to reference the exported `ContactModel` class. Our `beforeEach` function then creates an instance of the `ContactModel` class, passing a JavaScript object with the `Name` and `EmailAddress` properties into the constructor.



We are using JSON syntax in the constructor of our `ContactModel` class. This syntax closely matches the data that a RESTful service would return, and is, therefore, a handy way of constructing classes and assigning properties in a single constructor call.

Our first test is checking whether the `contactModel.Name` ES5 syntax works correctly, and will return the text `'testName'`. The second test is almost the same but uses the `.get('Name')` internal Backbone attribute syntax in order to ensure that our TypeScript class and the Backbone class are working as expected.

The require.config file

Now that we have defined a `Backbone.Model`, and have written a Jasmine test for it, we will need to run this test in a browser to verify our results. Generally, we would create an HTML page, and then include the `<script>` tags for each of our JavaScript files in the header section. This is where AMD steps in. We no longer need to specify every JavaScript file in our HTML. All we need to do is include a single `<script>` tag for Require (which is our module loader), which will then co-ordinate the loading of all the files that we need automatically.

To do this, let's create a `SpecRunner.html` file in the `/tests` directory as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>AMD SpecRunner</title>
  <link rel="stylesheet"
        type="text/css"
        href="/Scripts/jasmine/jasmine.css">
  <script
    data-main="/tscode/tests/TestConfig"
    type="text/javascript"
    src="/Scripts/require.js">
  </script>
</head>
<body>
</body>
</html>
```

This is a very simple HTML file. The line to note here, though, is the `<script>` tag that loads `/Scripts/require.js`. This script tag has a `data-main` attribute, which is set to `/tscode/tests/TestConfig`. The `data-main` attribute is passed to `Require`, and it tells `Require` where to start looking for our JavaScript files. In the preceding code, `Require` will look for a file named `/tscode/tests/TestConfig.js`.

We will build this `/tscode/tests/TestConfig.ts` file as follows:

```
require.config(
  {
    baseUrl: "../..",
    paths: {
      'jasmine': '/Scripts/jasmine/jasmine',
      'jasmine-html': '/Scripts/jasmine/jasmine-html',
      'jasmine-boot': '/Scripts/jasmine/boot',
      'underscore' : '/Scripts/underscore',
      'backbone': '/Scripts/backbone',
      'jquery': '/Scripts/jquery-2.1.1',
    },
    shim: {
      underscore: {
        exports: '_'
      },
      backbone : {
        deps: ['underscore'],
        exports: 'Backbone'
      },
      'jasmine' : {
        exports: 'window.jasmineRequire'
      },
      'jasmine-html': {
        deps : ['jasmine'],
        exports: 'window.jasmineRequire'
      },
      'jasmine-boot': {
        deps : ['jasmine-html', 'backbone'],
        exports: 'window.jasmineRequire'
      }
    }
  }
);
```

```
var specs = [
  'tscode/tests/models/ContactModelTests'
];

require(['jasmine-boot'], (jb) => {
  require(specs, () => {
    (<any>window).onload();
  });
});
```

We start with a call to the `require.config` function, and pass it a JavaScript object which has three properties: `baseUrl`, `paths`, and `shim`. The `baseUrl` property tells Require what base directory to use when it is looking for JavaScript files. In the sample application, our `TestConfig.ts` file is in the `/tscode/tests` directory, so our base directory would be `.`.

The `paths` property specifies the full path to our JavaScript files, and each entry is given a name. In the previous example, the script `/Scripts/jasmine/jasmine.js` is named `'jasmine'`, and can be referred to as `'jasmine'` throughout the rest of the script.



Require will automatically append `.js` to each of these entries, so any entry in the `paths` property should NOT include `.js` in the file's entry.

The `shim` property tells Require a few more details about each entry in the `paths` property. Take a look at the `shim` entry for `backbone`. It has a `deps` property that specifies what the dependencies for Backbone are. Backbone has a dependency on Underscore, so Underscore must be loaded before Backbone.

The `exports` property tells Require to append the library to the namespace that is specified as the `exports`' value. In our preceding sample, therefore, any call to Underscore must prepend an `_` to any function call in the Underscore library. As an example, `_.bindAll` calls the `bindAll` function of Underscore.

Dependencies specified in the `shim` section of `require.config` are recursive. If we take a look at the `shim` for `'jasmine-boot'`, we can see that it is dependent on `'jasmine-html'`, which in turn is dependent on `'jasmine'`. Require will ensure that all these scripts are loaded in the correct order, before running code that needs `'jasmine-boot'`.

Let's next take a look at the bottom of the file where we call the `require` function. This call takes two parameters: an array of files that need to be loaded, and a callback function to call once the load step has been completed. This callback function has a corresponding parameter for each of the file entries in our array. So, in the previous example, `'jasmine-boot'` will be made available to our function via the corresponding parameter `jb`. We will see more examples of this a bit later.

Calls to the `require` function, each with its array of files that need to be loaded, and the corresponding callback parameters, can be nested. In our sample, we have nested a second call to the `require` function inside our initial call, but this time we have passed in the `specs` array and omitted the callback parameters. This `specs` array currently contains just our `ContactModelTests` file. Our nested anonymous function just calls the `window.onload` function, which will trigger Jasmine to run all of our tests.



The call to `window.onload()` has a slightly strange syntax. We are using an explicit cast to cast the `window` variable to a type of `<any>` before calling the `onload()` function. This is because the TypeScript compiler is expecting an `Event` parameter to be passed to the `onload()` function. We do not have an event parameter, and need to ensure that the generated JavaScript is in the correct syntax - hence the cast to `<any>`.

If all goes well, we can now fire up our browser and call the `SpecRunner.html` page at `/tscode/tests/SpecRunner.html`.

Fixing Require config errors

Quite often, when developing AMD applications with `Require`, we can start to get unexpected behaviour, strange error messages, or simply blank pages. These strange results are generally caused by the configuration for `Require`, either in the `paths`, `shim`, or `deps` properties. Fixing these AMD errors can be quite frustrating at first, but generally, they are caused by one of two things – incorrect dependencies or `file-not-found` errors.

To fix these errors, we will need to open the debugging tools within the browser that we are using – which for most browsers, is achieved by simply hitting `F12`.

Incorrect dependencies

Some AMD errors are caused by incorrect dependencies in our `require.config`. These errors can be found by checking the console output in the browser. Dependency errors would generate browser errors similar to the following:

```
ReferenceError: jasmineRequire is not defined
```

```
ReferenceError: Backbone is not defined
```

This type of error might mean that the AMD loader has loaded Backbone, for example, before loading Underscore. So, whenever Backbone tries to use an underscore function, we get a `not defined` error, as shown in the preceding output. The fix for this type of error is to update the `deps` property of the library that is causing the error. Make sure that all prerequisite libraries have been named in the `deps` property, and the errors should go away. If they do not, then the error may be caused by the next type of AMD error, a `file-not-found` error.

404 errors

File-not-found, or 404 errors are generally indicated by console output similar to the following:

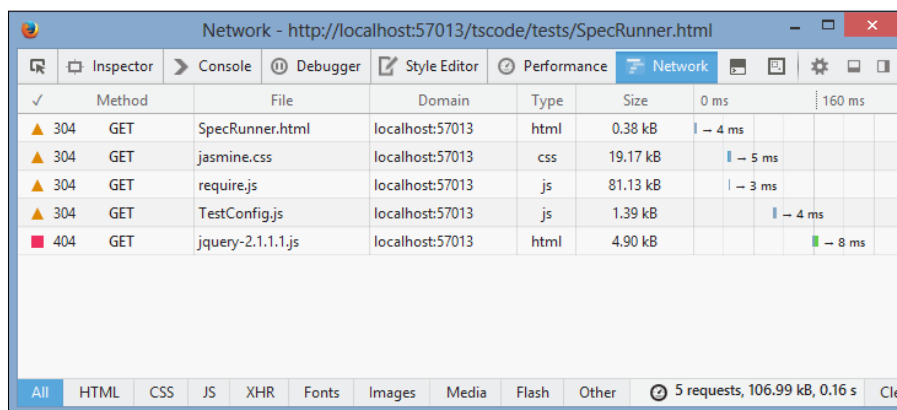
```
Error: Script error for: jquery
```

```
http://requirejs.org/docs/errors.html#scripterror
```

```
Error: Load timeout for modules: jasmine-boot
```

```
http://requires.org/docs/errors.html#timeout
```

To find out which file is causing the preceding error, switch to the network tab in your debugger tools and refresh the page. Look for 404 (`file-not-found`) errors, as shown in the following screenshot:



Firefox network tab with 404 errors

In this screenshot, we can see that the call to `jquery.js` is generating a 404 error, as our file is actually named `/Scripts/jquery-2.1.1.js`. These sorts of errors can be fixed by adding an entry to the `paths` parameter in `require.config` so that any call to `jquery.js` is replaced by a call to `jquery-2.1.1.js`.



Require has a good set of documentation for common AMD errors (<http://requirejs.org/docs/errors.html>) as well as advanced API usages, including circular references (<http://requirejs.org/docs/api.html#circular>), so be sure to check the site for more information on possible AMD errors.

Using Backbone.Collections

Now that we have a `ContactModel` working and tested, we can build a `Backbone.Collection` to house a group of `ContactModel` instances. Since we are using AMD, we can create a new `ContactCollection.ts` file and add the following code:

```
import cm = require("../ContactModel")
export class ContactCollection
  extends Backbone.Collection<cm.ContactModel> {
  model = cm.ContactModel;
  url = "/tscode/tests/contacts.json";
}
```

Creating a `Backbone.Collection` is relatively straightforward. Firstly, we import the `ContactModel`, as we have seen previously, and assign it to the `cm` namespace. We then create a class named `ContactCollection` that extends from `Backbone.Collection`, and uses the generic type `cm.ContactModel`. This `ContactCollection` has two properties: `model` and `url`. The `model` property tells Backbone what model class to use internally, and the `url` property points to a server-side RESTful URL. Backbone will generate the correct POST, GET, DELETE, and UPDATE HTTP protocols for server-side RESTful calls when we synchronize our data with the server. In the preceding sample, we are simply returning a hardcoded JSON file, as we will only be using HTTP GETs.

If we open the resultant JavaScript file that TypeScript generates, we will see that the compiler has modified our file quite a bit:

```
var __extends = this.__extends || function (d, b) {
  for (var p in b) if (b.hasOwnProperty(p)) d[p] = b[p];
  function __() { this.constructor = d; }
  __.prototype = b.prototype;
```

```
    d.prototype = new __();
  };
  define(["require", "exports", "./ContactModel"], function
    (require, exports, cm) {
    var ContactCollection = (function (_super) {
      __extends(ContactCollection, _super);
      function ContactCollection() {
        _super.apply(this, arguments);
        this.model = cm.ContactModel;
        this.url = "/tscode/tests/contacts.json";
      }
      return ContactCollection;
    })(Backbone.Collection);
    exports.ContactCollection = ContactCollection;
  });
  //# sourceMappingURL=ContactCollection.js.map
```

The first six lines of the file starting with `var __extends`, are simply TypeScript's implementation of inheritance in JavaScript, and we will not concern ourselves too much with it.

The lines to note start with the `define` function. TypeScript has wrapped our class definition within an outer call to `define`. This `define` function call now has three parameters: `require`, `exports`, and `./ContactModel`. The syntax and usage of this function are exactly the same as the call to the `require` function that we wrote ourselves in the `TestConfig.ts` file.

The first parameter is an array of files to import, and the second parameter is a callback function to call once these files have been loaded. Again, each element in our first array has a corresponding argument in our callback parameters. TypeScript will automatically add the `"require"` and `"exports"` parameters for us, and then include any file that we specified using the `import` keyword. When TypeScript compiles our files using the AMD syntax, it will automatically generate this style of JavaScript to be compatible with AMD loaders such as Require.

Let's now write a couple of unit tests for our `ContactCollection`:

```
import cc = require("../..../app/models/ContactCollection");
import cm = require("../..../app/models/ContactModel");
describe("/tests/models/ContactCollectionTests", () => {
  it("should create a collection", () => {
    var contactCollection = new cc.ContactCollection(
    [
```

```

        new cm.ContactModel(
            { Name: 'testName1', EmailAddress: 'testEmail1' } ),
        new cm.ContactModel(
            { Name: 'testName2', EmailAddress: 'testEmail2' } )
    ] );
    expect( contactCollection.length ).toBe( 2 );
    });
});

```

This test starts with an `import` statement for both the `ContactCollection`, as well as the `ContactModel`, as we will be using both within this test. It then simply creates a new `ContactCollection` and passes in an array of two new `ContactModels`. This test highlights how to create a new `ContactCollection`, and populate it programmatically.

Let's now write a test to load the collection via the `url` property:

```

describe("contact json tests", () => {
    var collection: cc.ContactCollection;
    it("should load collection from url", () => {
        collection = new cc.ContactCollection();
        collection.fetch({ async: false });
        expect( collection.length ).toBe( 4 );
    });
});

```

This test creates a new `ContactCollection` and then calls the `fetch` function.



We have passed an `async` flag set to `false` to force Backbone to use a synchronous call to the server. In other words, the JavaScript will pause until the `fetch` is complete before moving onto the next line. We could have written this test using the asynchronous `done` syntax of Jasmine, but for smaller tests, passing this `async` flag makes the code a little easier to read.

As mentioned previously, the `fetch` function will use the `url` parameter to issue a GET HTTP request to the provided URL, which in this case is simply loading the `contacts.json` file. The contents of this file are as follows:

```

[
  { "Name": "Mr Test Contact",
    "EmailAddress": "mr_test_contact@test.com" },
  { "Name": "Mrs Test Contact",

```



```
      "EmailAddress": "mrs_test_contact@test.com" },
    { "Name": "Ms Test Contact",
      "EmailAddress": "ms_test_contact@test.com" },
    { "Name": "Dr Test Contact",
      "EmailAddress": "dr_test_contact@test.com" }
  ]
```

This file uses simple JSON syntax to define four contacts, each with a `Name` and `EmailAddress` property. Let's write a few integration tests to ensure that the `fetch` function, using this JSON, actually creates a `ContactCollection` correctly:

```
describe("contact json model tests", () => {
  var collection: cc.ContactCollection;
  beforeEach(() => {
    collection = new cc.ContactCollection();
    collection.fetch({ async: false });
  });
  it("ContactModel at 0 should have attribute called Name", () => {
    var contactModel = collection.at(0);
    expect(contactModel.get('Name')).toBe('Mr Test Contact');
  });
  it("ContactModel at 0 should have property called Name", () => {
    var contactModel : cm.ContactModel = collection.at(0);
    expect(contactModel.Name).toBe('Mr Test Contact');
  });
});
```

In this test code, we are using the `beforeEach` function to populate our collection variable with an instance of the `ContactCollection` class, and are then calling the `fetch` function, again with the `{async: false}` flag. Our first test then uses the Backbone `at` function to retrieve the first model held within the collection at index 0. We then check the `'Name'` attribute of the returned model, using Backbone's internal `get` function. The second test is using the ES5 syntax of our `ContactModel` class, just to test whether Backbone is in fact storing an instance of our `ContactModel` class in its collection.

To include these tests in our test suite, we now simply need to modify the `TestConfig.ts` file and add an entry to our `specs` array as follows:

```
var specs = [
  'tscode/tests/models/ContactModelTests',
  'tscode/tests/models/ContactCollectionTests'
];
```

Backbone views

Now that we have a `ContactCollection` to house our `ContactModels`, let's create a `Backbone.View` that will render this collection to the DOM. In order to do this, we will actually create two views: one view for each item in the collection, and one view for the collection itself. Remember that Backbone views combine a `Backbone.Model` with a template in order to render the model's properties into the DOM.


We will start with the view to render a single collection item (in this case a single `ContactModel`), called `ContactItemView`:

```
import cm = require("../models/ContactModel");
export class ContactItemView extends Backbone.View<cm.ContactModel> {
  template: (properties?: any) => string;
  constructor(options?: any) {
    this.className = "contact-item-view";
    this.template = _.template(
      '<p><%= Name %> (<%= EmailAddress %>)</p>');
    super(options);
  }
  render(): ContactItemView {
    this.$el.html(this.template(this.model.attributes));
    return this;
  }
}
```

This code snippet starts with an `import` of the `ContactModel` class that we have attached to the `cm` namespace. We then create a class named `ContactItemView` that extends from `Backbone.View`. Similar to the generic syntax that we used for our collection, this view class also uses the `ContactModel` as the type for its generic instance. Finally, we export this class to make it available to our code as an AMD module.

The `ContactItemView` class has a public property named `template` that is a function that returns a string. This function takes the model's properties as an input argument. The `template` function is assigned in the second line of the constructor, to be the result of the call to Underscore's `_.template(...)` function. If we take a closer look at the string used in this template function, we will see that it is an HTML string that uses the `<%= propertyName %>` syntax, to inject the Backbone model's properties into the HTML. We have also specified that the DOM `className` should be set to `"contact-item-view"`. Finally, we call the base class constructor with the `options` argument that was passed into the constructor.

So, what have we done here? We have created a `Backbone.View` class, specified its `className`, and set the `template` that the view should use to render its model to the DOM. The last piece of code that we need is the `render` function itself. This `render` function does a couple of things in just one line. Firstly, each Backbone view has a `$el` property that holds the DOM element. We then call the `html` function on this element in order to set its HTML, and pass in the result of a call to the `template` function. By convention, the `render` function always returns `this`, to enable a calling class to use fluent syntax after calling the `render` function.

 There are a number of template engines that can be used with Backbone—such as Handlebars (<http://handlebarsjs.com/>) and Moustache (<https://github.com/janl/mustache.js/>) to name a few. In this sample, we will just stick to the Underscore template engine.

Now that we have a `Backbone.View` defined, we can write a simple test for it:

```
import cm = require("../app/models/ContactModel");
import ccv = require("../app/views/ContactItemView");
describe("/tscode/tests/views/ContactItemViewTests", () => {
  it("should generate html from template and model", () => {
    var contactModel = new cm.ContactModel(
      { Name: 'testName', EmailAddress: 'testEmailAddress' });

    var contactItemView = new ccv.ContactItemView(
      { model: contactModel });
    var html = contactItemView.render().$el.html();

    expect(html).toBe('<p>testName (testEmailAddress)</p>');
  });
});
```

This code snippet starts with the imports for both `ContactModel` and `ContactItemView`. There is only one test in this suite, and it is fairly simple. Firstly, we create an instance of a `ContactModel`, setting the `Name` and `EmailAddress` properties in the constructor. We then create an instance of the `ContactItemView` class, and pass the model we just created as a constructor argument. Note the syntax that we are using in the constructor: `{ model: contactModel }`. Backbone views can be constructed in a few different ways, and the properties that we set on construction – in this case the `model` property – are passed down to the base Backbone classes, via the `super()` function call in our constructor.

Our test then calls the `render` function on the `contactItemView` instance. Note here that we are then referencing the `$el` property of the view directly, and calling the `html` function – as if it were a jQuery DOM element. This is the reason why all render functions should return `this`.

Our test then checks that the result of the `render` function generates the HTML that we expect, based on the template, and our model properties.

Using the Text plugin

Having hardcoded HTML within our view, however, will make our code difficult to maintain. To help with this conundrum, we will use a Require plugin called Text. Text uses normal require syntax, just with a `'text!'` prefix to load files from the site for use in our code. To install this plugin via NuGet, simply type:

```
Install-package RequireJS.Text
```

To use Text, we will first need to list `text` in our `require.config` `paths` property as follows:

```
paths: {
  // existing code
  'text': '/Scripts/text'
},
```

We can then modify our call to `require` in our `TestConfig.ts` as follows:

```
var CONTACT_ITEM_SNIPPET = "";
require(
  ['jasmine-boot',
   'text!/tscode/app/views/ContactItemView.html'],
  (jb, contactItemSnippet) => {
    CONTACT_ITEM_SNIPPET = contactItemSnippet;
    require(specs, () => {
      (<any>window).onload();
    });
  });
```

In this code snippet, we create a global variable named `CONTACT_ITEM_SNIPPET` to hold our snippet, and then we include the HTML file that we need to load using the `'text!<path to html>'` syntax in our call to `require`. Again, each item in the array we use for the `require` function call has a corresponding variable in our anonymous function.

In this way, Require will load the text found at `/tscode/app/views/ContactItemView.html`, and pass it to our function via the `contactItemSnippet` argument as a string. We can then set the global variable `CONTACT_ITEM_SNIPPET` to this value. Before we can run this code, however, we will need to modify our `ContactItemView` to use this variable:

```
constructor(options?: any) {
  this.className = "contact-item-view";
  this.events = <any>{ 'click': this.onClicked };
  this.template = _.template(CONTACT_ITEM_SNIPPET);

  super(options);
}
```

The changed line in the preceding code is the call to invoke the `_.template` function using the value of the global variable `CONTACT_ITEM_SNIPPET`, instead of a hard coded HTML string.

The last thing we need is to create the `ContactItemView.html` file itself, as follows:

```
<div class="contact-outer-div">
  <div class="contact-name-div">
    <%= Name %>
  </div>
  <div class="email-address-div">
    (<%= EmailAddress %>)
  </div>
</div>
```

This HTML file uses the same `<%= propertyName %>` syntax that we have seen before, but we are now able to easily expand our HTML to include outer divs, and give each property its own CSS classes for some styling later on.

Running our tests now, however, will break our `ContactItemViewTests` – because the HTML we are using has been changed. Let's fix this broken test now:

```
//expect(html).toBe('<p>testName (testEmailAddress)</p>');
expect(html).toContain('testName');
expect(html).toContain('testEmailAddress');
```

We have commented the offending line, and are using the `.toContain` matcher to ensure that our HTML has been injected correctly with the model properties, instead of looking for an exact match for the `html` string value.

Rendering a collection

Now that we have a view to render individual Contact items, we need another view to render the entire ContactCollection. To do this, we simply create a new Backbone.View for our collection, and then create a new ContactItemView instance for each item in the collection as follows:

```
import cm = require("../models/ContactModel");
import civ = require("../ContactItemView");
export class ContactCollectionView extends
  Backbone.View<Backbone.Model> {
  constructor(options?: any) {
    super(options);
    _.bindAll(this, 'renderChildItem');
  }

  render(): ContactCollectionView {
    this.collection.each(this.renderChildItem);
    return this;
  }

  renderChildItem(element: Backbone.Model, index: number) {
    var itemView = new civ.ContactItemView( { model: element } );
    this.$el.append(itemView.render().$el);
  }
}
```

We start this code snippet with our imports for the ContactModel and ContactItemView modules. We then create a ContactCollectionView that extends Backbone.View, this time using a base Backbone.Model for the generic syntax. Our constructor simply passes any options that it receives down to the base view class through the super function call. We then call an Underscore function named bindAll. The Underscore bindAll function is a utility function that binds the scope of this to the correct context, when used in a class function. Let's explore the code a little to make this clearer.

The render function will be called by the user of the ContactCollectionView, and simply calls the renderChildItem function for each model that it has in its collection. this.collection.each takes a single parameter, which is a callback function to be called for each model in the collection. We could have written this code as follows:

```
render(): ContactCollectionView {
  this.collection.each(
    (element: Backbone.Model, index: number) => {
  // include rendering code within this anonymous function
    }
  );
  return this;
}
```

This version of the same code uses an anonymous function within the each function. In our previous code snippet, however, we have written the `renderChildItem` as a class function, instead of using an anonymous function. Because of JavaScript's lexical scoping rules, this slight change means that the `this` property would now refer to the function itself, and not the class instance. By using `_.bindAll(this, 'renderChildItem')`, we have bound the variable `this` to be the class instance for all calls to `renderChildItem`. We can then use the `this` variable within the `renderChildItem` function, and `this.$el` will be correctly scoped to the instance of the class `ContactCollectionView`.

Now for a couple of tests on this `ContactCollectionView` class:

```
import cc = require("../..app/models/ContactCollection");
import cm = require("../..app/models/ContactModel");
import ccv = require("../..app/views/ContactCollectionView");
describe("/ts/views/ContactCollectionViewTests", () => {
  var contactCollection: cc.ContactCollection;
  beforeEach(() => {
    contactCollection = new cc.ContactCollection([
      new cm.ContactModel(
        { Name: 'testName1', EmailAddress: 'testEmail1' } ),
      new cm.ContactModel(
        { Name: 'testName2', EmailAddress: 'testEmail2' } )
    ]);
  });

  it("should create a collection property on the view", () => {
    var contactCollectionView = new ccv.ContactCollectionView({
      collection: contactCollection
    });
    expect(contactCollectionView.collection.length).toBe(2);
  });
});
```

In this code snippet, the `import` and `beforeEach` functions should be pretty easy to decipher, so let's focus on the body of the actual test. Firstly, we are creating a `ContactCollectionView` instance, and passing in this `contactCollection` instance via the `{ collection: contactCollection }` property in the constructor. Backbone views that work with a single item use the `{ model: <modelName> }` property, and views that work with collections use the `{ collection: <collectionInstance> }` property. Our first test simply checks to see that the internal `collection` property does actually contain a collection whose `length` should be 2.

We can now write a test to check that the `renderChildItem` function is called when we call the `render` function on our `ContactCollectionView` as follows:

```
it("should call render on child items", () => {
    var contactCollectionView = new ccv.ContactCollectionView({
        collection: contactCollection
    });
    spyOn(contactCollectionView, 'renderChildItem');
    contactCollectionView.render();

    expect(contactCollectionView.renderChildItem).toHaveBeenCalled();
});
```

This test creates a view as we have seen previously, and then creates a spy on the `renderChildItem` function. To trigger this function to be called, we call the `render` function on our view instance. Finally, we just check that our spy has been called.

Next, we can write a quick test to see if the HTML generated by the `render` function contains properties from our collection's models:

```
it("should generate html from child items", () => {
    var contactCollectionView = new ccv.ContactCollectionView({
        collection: contactCollection
    });
    var renderedHtml = contactCollectionView.render().$el.html();
    expect(renderedHtml).toContain("testName1");
    expect(renderedHtml).toContain("testName2");

});
```

This test is very similar to our `ContactItemView` rendering tests, but instead uses the `ContactCollectionView` `render` function.

Creating an application

With the two Backbone views in place, we can now build a simple class to coordinate the loading of our collection, and the rendering of the full collection to the DOM:

```
import cc = require("tscode/app/models/ContactCollection");
import cm = require("tscode/app/models/ContactModel");
import civ = require("tscode/app/views/ContactItemView");
import ccv = require("tscode/app/views/ContactCollectionView");
export class ContactViewApp {
  run() {
    var contactCollection = new cc.ContactCollection();
    contactCollection.fetch(
      {
        success: this.contactCollectionLoaded,
        error: this.contactCollectionError
      }
    );
  }

  contactCollectionLoaded(model, response, options) {
    var contactCollectionView = new ccv.ContactCollectionView(
      {
        collection: model
      }
    );
    $("#mainContent").append(
      contactCollectionView.render().$el;
    )
  }
  contactCollectionError(model, response, options) {
    alert(model);
  }
}
```

Our code starts with imports for each of our various modules. We then create a class definition named `ContactViewApp`, and within this class, a method named `run`. This `run` method simply creates a new `ContactCollection`, and calls `fetch` to trigger Backbone to load the collection. This call to `fetch` then defines a success and error callback, each set to their relevant functions within the class.

When the `ContactCollection` `fetch` returns successfully, Backbone will invoke the `contactCollectionLoaded` function. Within this function, we simply create a `ContactCollectionView`, and then use jQuery to append the HTML returned via the `render` function to the DOM element `"#mainContent"`.

We can now create a web page to put everything together. The contents of our HTML page would now read as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
<head>
  <title>Contacts View</title>
  <link rel="stylesheet" type="text/css"
        href="/css/app.css">
  <script data-main="/tscode/app/AppConfig"
          type="text/javascript"
          src="/Scripts/require.js"></script>

</head>
<body>
  <div id="mainContent"></div>
</body>
</html>
```

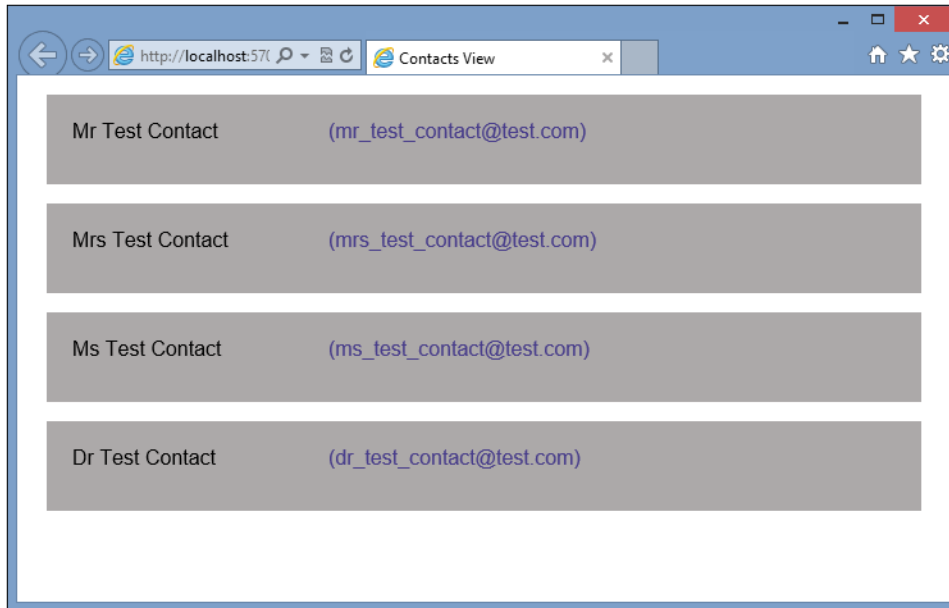
This page is very similar to the page we were using previously for running our tests. We have included an `app.css` link to allow for some styling, and we then call `Require` with a new config file, named `/tscode/app/AppConfig`. We also have a `div` within the body tag, with an id of `mainContent`, which will house the rendered HTML from our `ContactViewApp`. Now we need to create our `AppConfig.ts` file for `Require` to use, as follows:

```
require.config(
  {
    baseUrl: "../..",
    paths: {
      'underscore': '/Scripts/underscore',
      'backbone': '/Scripts/backbone',
      'jquery': '/Scripts/jquery-2.1.1',
      'ContactViewApp': '/tscode/app/ContactViewApp',
      'text': '/Scripts/text'
    },
    shim: {
      underscore: {
        exports: '_'
      },
      backbone: {
        deps: ['underscore'],
        exports: 'Backbone'
      }
    },
    ContactViewApp: {
      deps: ['backbone']
    }
  }
)
```

```
);  
  
var CONTACT_ITEM_SNIPPET = "";  
  
require([  
  'ContactViewApp',  
  'text!/tscode/app/views/ContactItemView.html'  
], (app, contactItemSnippet) => {  
  
  CONTACT_ITEM_SNIPPET = contactItemSnippet;  
  var appInstance = new app.ContactViewApp();  
  appInstance.run();  
});
```

The first thing to note in this code snippet, is that we have now included a `paths` reference to our `ContactViewApp`. The corresponding shim entry for `ContactViewApp` specifies that it has a dependency on `backbone`. Again, we have a global variable named `CONTACT_ITEM_SNIPPET`, and we then call the `require` function to load our `ContactViewApp` class, as well as the HTML snippet. Note too, that we are able to reference our `ContactViewApp` via the `app` argument in our anonymous function, and the HTML via the `contactItemSnippet` argument. To run the app, we simply create an instance of the `ContactViewApp` class, and call the `run` method.

We should now be able to see the results of all of our hard work:



The Backbone app running with Require.js

Using jQuery plugins

To finish off our app, let's use a jQuery plugin called **flip** (<http://lab.smashup.it/flip/>) that triggers an animation to rotate, or flip, the outer `div` of an item when it is clicked. Flip is typical of a range of jQuery plugins that can be applied to elements of our application. Before we can trigger a Flip animation, however, we will need to respond to a click event from the user within the `ContactItemView` as follows:

```
import cm = require("../models/ContactModel");

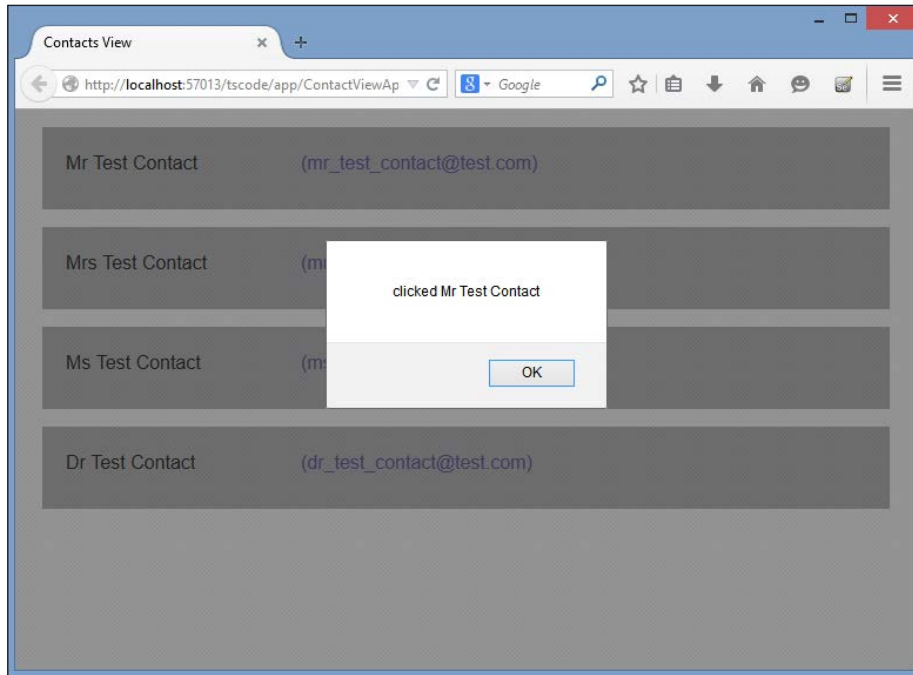
export class ContactItemView extends Backbone.View<cm.ContactModel> {
  template: (properties?: any) => string;
  constructor(options?: any) {
    this.className = "contact-item-view";
    this.events = <any>{ 'click': this.onClicked };
    this.template = _.template(CONTACT_ITEM_SNIPPET);
    super(options);
  }

  render(): ContactItemView {
    this.$el.html(this.template(this.model.attributes));
    return this;
  }

  onClicked() {
    alert('clicked : ' + this.model.Name);
  }
}
```

In this code snippet, we have now added an `onClicked` function to our `ContactItemView` class that simply pops up an alert. Note how we are able to reference the `model` property of the view class, in order to read properties from the underlying `Backbone.Model` that this class instance was created with. Within the constructor, we have also set `this.events` to a JavaScript object that has a single property: `'click'`.

The 'click' property is set to our onCliked function, and will be invoked when the ContactItemView DOM element receives a user's click event. With this in place, whenever we click on a rendered element in our page, we will receive an alert popup:



Alert popup on click event showing Model properties

We can now turn our attention to using the Flip jQuery plugin. Flip relies on jQuery as well as jQueryUI, so we will need to install jQueryUI from NuGet as follows:

Install-package jQuery.UI.Combined

Flip itself does not have a NuGet package, so will need to download it, and included it in our project the old-fashioned way. There is also no DefinitelyTyped definition for Flip, so we will need to include one in our project as follows:

```
interface IFlipOptions {
    direction: string;
    onBefore?: () => void;
    onAnimation?: () => void;
    onEnd?: () => void;
    speed?: number;
    color?: string;
    content?: string;
}
```

```

interface JQuery {
  flip(input: IFlipOptions): JQuery;
  revertFlip();
}

```

This declaration file for the Flip plugin is very simply generated from the documentation on the website. As Flip is a jQuery plugin, it is available on any jQuery object that is reference by the `$()` notation. Because of this, we must extend the JQuery type definition with our own – hence we create the JQuery interface with our two new functions: `flip` and `revertFlip`. The input to Flip has been defined as the `IFlipOptions` interface, as built from the website documentation.

To load this library within Require, we modify our call to `require.config` as follows:

```

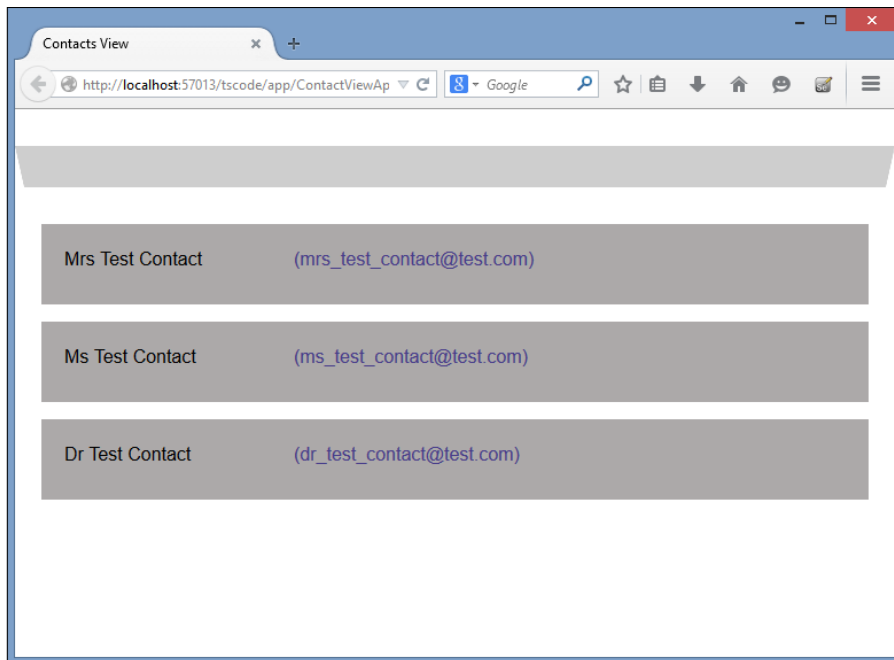
require.config(
  {
    baseUrl: "../..",
    paths: {
      'underscore': '/Scripts/underscore',
      'backbone': '/Scripts/backbone',
      'jquery': '/Scripts/jquery-2.1.1',
      'ContactViewApp': '/tscode/app/ContactViewApp',
      'text': '/Scripts/text',
      'jqueryui': '/Scripts/jquery-ui-1.11.2',
      'jqueryflip' : '/Scripts/jquery.flip'
    },
    shim: {
      underscore: {
        exports: '_'
      },
      backbone: {
        deps: ['underscore'],
        exports: 'Backbone'
      },
      jqueryui: {
        deps: ['jquery']
      },
      jqueryflip: {
        deps: ['jqueryui'],
        exports: '$'
      },
      ContactViewApp: {
        deps: ['backbone', 'jqueryflip']
      }
    }
  }
);

```

Here, we have added two entries to our paths object: `jqueryui`, and `jqueryflip`. We have then added corresponding `shim` entries and specified the relevant dependencies. The line to note here, is the `exports` property on `jqueryflip`. We have specified that it must export to the `$` symbol. This is the default jQuery selector symbol, and all jQuery plugins must export to the `$` symbol, in order to be defined correctly when using Require. Our final change to the code is to use the `flip` function on the click event of `ContactItemView` as follows:

```
onClicked() {  
  this.$el.flip({  
    direction: 'tb',  
    speed : 200  
  });  
}
```

Here we are referencing the `$el` element within the `Backbone.View`, which is a shorthand syntax for the jQuery selector. We are then calling the `flip` function, and specifying a top-to-bottom flip, to last 200 milliseconds. Running our page now, and clicking on a contact element will now trigger a flip animation:



Flip.js in action flipping a div element

Summary

In this chapter we have had a look at using modules - both CommonJs and AMD. We explored CommonJS modules as used within Node applications, and discussed the creation and usage of these modules with TypeScript. We then moved on to browser-based modules, and explored the use of AMD compilation in regards to Require. We built a very simple Backbone based application, complete with Jasmine unit tests, and then had a look at using the Text plugin with Require. We also incorporated a third-party jQuery plugin called Flip to provide some animation on our user interface. In our next chapter, we will tackle some object-oriented programming principles, and have a look at dependency injection and domain events.

8

Object-oriented Programming with TypeScript

In 1995, the **Gang of Four (GoF)**, published a book named *Design Patterns: Elements of Reusable Object-Oriented Software*. In it, the authors, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, describe a number of classic software design patterns. These patterns present simple and elegant solutions to common software problems. If you have never heard of design patterns such as the Factory pattern, Composite pattern, Observer pattern, or Singleton pattern, then going through this GoF book is highly recommended.

The Design patterns presented by the GoF have been reproduced in many different programming languages, including Java and C#. Mark Torok has even ported these patterns to TypeScript, and his GitHub repository can be found at https://github.com/torokmark/design_patterns_in_typescript. We have already explored one of these patterns, the Factory Design Pattern, in *Chapter 3, Interfaces, Classes and Generics*, and Mark's work provides a quick and simple reference implementation of all of the GoF patterns in TypeScript.

Simon Timms has also published a book called *Mastering JavaScript Design Patterns*, Packt Publishing (<https://www.packtpub.com/application-development/mastering-javascript-design-patterns>), which walks the reader through each of these patterns, when to use them, and how to use them.

In this chapter, we will not cover the standard GoF design patterns, but instead take a look at two other popular design patterns and how they can be implemented in TypeScript. We will discuss Dependency Injection using a Service Locator Pattern, and then see how these techniques can be used to build a Domain Event Pattern implementation.

Program to an interface

One of the primary notions that the GoF adhere to, is the idea that programmers should "program to an interface, and not an implementation". This means that programs are built using interfaces as the defined interaction between objects. By programming to an interface, client objects are unaware of the internal logic of their dependent objects, and are much more resilient to change.

The TypeScript language brings with it the `interface` keyword, allowing us to write object-oriented code against interfaces in a much easier way than with standard JavaScript. Remember, though, that interfaces are a TypeScript concept only, and are compiled away in the generated JavaScript.

Note that many other languages have the concept of being able to interrogate an object to see which interfaces they implement, a process called **reflection**.

SOLID principles

An extension of the "program to an interface" principle, is what has been coined as SOLID design principles, based on the ideas of Robert Martin. This is an acronym for five different programming principles, and deserves a mention whenever object-oriented programming is discussed. Each of the letters in the word SOLID relate to an object-oriented principle, as follows:

- S: Single Responsibility
- O: Open Closed
- L: Liskov Substitution
- I: Interface Segregation
- D: Dependency Inversion

Single Responsibility

The idea behind the Single Responsibility principle is that an object should have just a single responsibility, or a single reason to exist. In other words, do one thing and do it well. We have seen examples of this principle in the previous chapter, in our work with Backbone. A Backbone model class is used to represent a single model. A Backbone collection class is used to represent a collection of these models, and a Backbone view class is used to render models or collections.

Open Closed

The idea behind the Open Closed principle states that an object should be open to extension, but closed for modification. In other words, once an interface has been designed for a class, changes that occur over time to this interface, should be implemented through inheritance, and not by modifying the interface directly.

Note that if you are writing libraries that are consumed by third-parties via an API, then this principle is essential. Changes to an API should only be made through a new, versioned release, and should not break the existing API or interface.

Liskov Substitution

The Liskov Substitution principle states that if one object is derived from another, then these objects can be substituted for each other without breaking functionality. While this principle seems fairly easy to implement, it can get pretty hairy when dealing with subtyping rules that relate to more complex types, such as lists of objects or actions on objects – which are most often found in code that works with generics. In these instances, the concept of variance is introduced, and objects can be either covariant, contravariant, or invariant. We will not discuss the finer points of variance here, but keep this principle in mind when writing libraries or code using generics.

Interface Segregation

The idea behind the Interface Segregation principle is that many interfaces are better than one general-purpose interface. If we tie this principle in with the Single Responsibility principle, we will start to look at our interfaces as smaller pieces of the puzzle, which will be put together to create broader application functionality.

Dependency Inversion

The Dependency Inversion principle states that we should depend on abstractions (or interfaces) rather than instances of concrete objects. Again, this is the same principle as "program to an interface, and not an implementation".

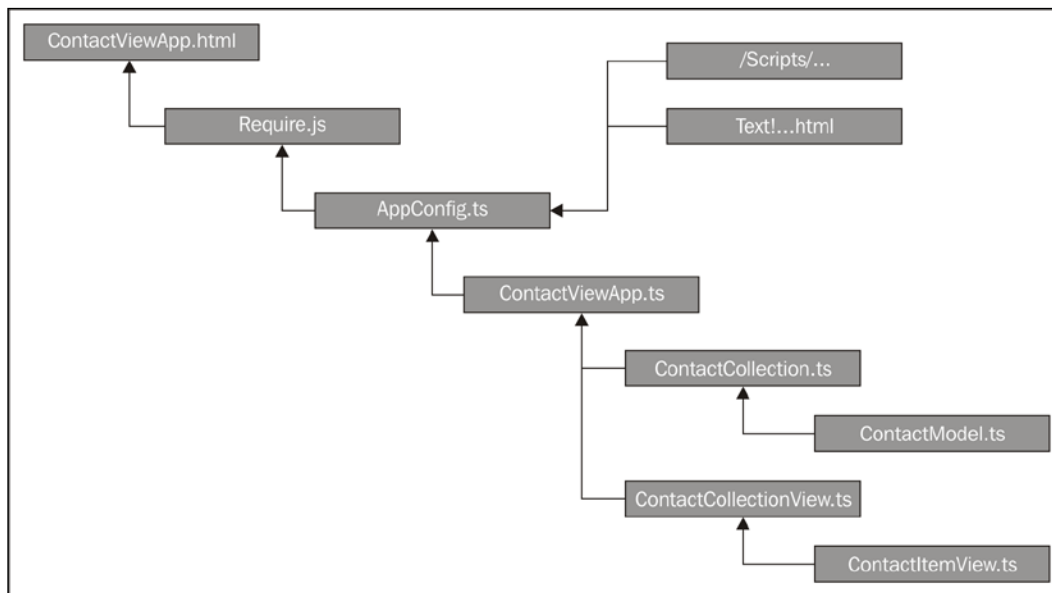
Building a Service Locator

The idea of the Service Location Pattern is that some areas of an application can be broken down into services. Each service should adhere to our SOLID design principles, and provide a small external interface that acts as a service API. Each service used by an application is then registered with a service locator. When a specific piece of information or functionality is required by the application, it can query this service locator to find the correct service, based on the service interface.

The problem space

In the previous chapter, we explored Backbone, where our application was broken down into models, collections, and views. Outside of these elements, we also had an application class to coordinate the loading of data via a collection, and the rendering of this collection using a view. Once our application classes were built, the last piece of the puzzle was putting together the `require.config` object, in order to coordinate the loading of our AMD modules, any HTML that we needed in our application, and our jQuery plugins.

If we have a look at a visual representation of which aspect of the application loaded which files, we come up with something that looks as follows:

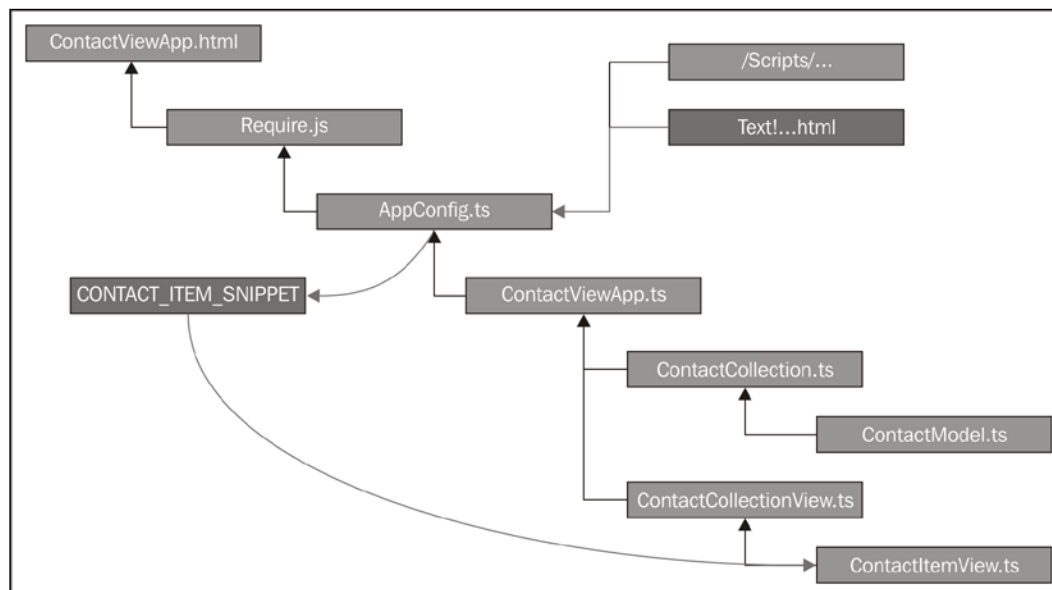


Application object dependency tree

We start at the top with an HTML page named `ContactViewApp.html`, which is the main entry page for our application, and which will be served up to the web browser. This HTML page then loads the `Require` library, which in turn loads our `AppConfig.ts` file containing the `require.config` section. This `require.config` section then instructs `Require` to load various scripts from the `/Scripts/` directory, as well as a snippet of HTML via the `Text` plugin. Once all files have been loaded by `Require`, the last portion of the `AppConfig.ts` file then loads the `ContactViewApp.ts`, which in turn loads our `ContactCollection.ts` and `ContactCollectionView.ts` files. These last two files then instruct `Require` to load the module files named `ContactModel.ts` and `ContactItemView.ts` respectively.

If we take a closer look at this hierarchy, it is quite feasible to imagine that in a large application, we would have a large amount of collections, models, views, and item views. It may be that we are loading collections of collections, and views containing sub-views that contain further sub-views. Each of these views will require some HTML to be loaded via the `Text` plugin, in order to use our template mechanism.

Let's take a closer look at how we loaded and used an HTML snippet in our previous example:



Dependency tree with usage of global variable

In this diagram, we can see that we loaded an HTML snippet via the `Text` plugin, within the `AppConfig.ts` file, and then stored it into a global variable named `CONTACT_ITEM_SNIPPET`. The only code that used this global variable was the `ContactItemView` class itself.

Using a global variable breaks our Dependency Inversion principle, in that we are programming to a concrete instance of a global variable, instead of an interface. This global variable can also be inadvertently changed by any running code, which may cause our views to stop functioning. Another problem that we faced when running our test suite, was that changing the original HTML template broke some of our unit tests. While we were able to modify the tests slightly in order to pass, this broken test highlighted that we had broken the Open Closed principle somewhere along the line.

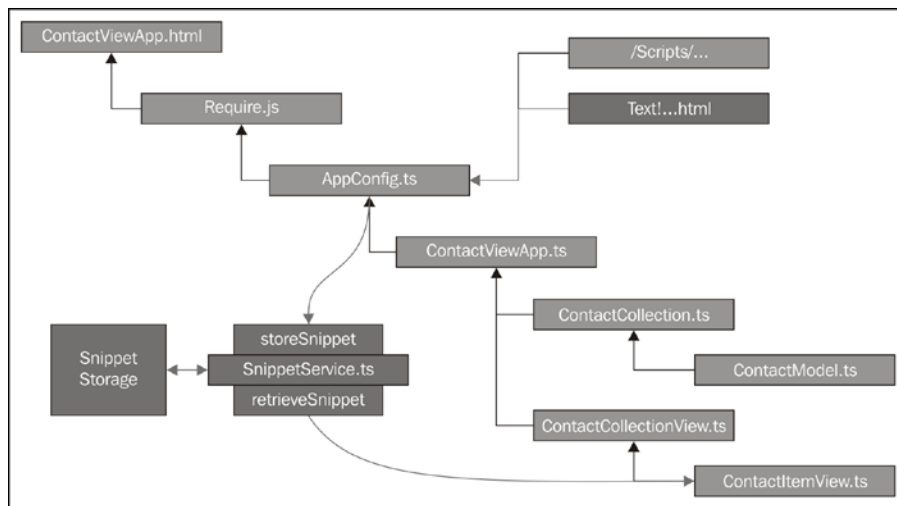
Creating a Service

We will solve the problem of using a global variable to store HTML snippets in two parts.

Firstly, let's define a service to replace our global variables – a `SnippetService`. This service will have a very simple interface, and will be only responsible for two things: storing an HTML snippet and retrieving an HTML snippet.

Secondly, we need a mechanism to get hold of this `SnippetService`, both at the point in our code where we store the snippet (in `AppConfig.ts`), and also at the point where we use the snippet (in `ContactItemView.ts`). We will use a Service Locator at both of these touch-points a bit later, but for now, let's flesh out a design for our snippet service.

Introducing a `SnippetService` changes our dependency diagram as follows:



Dependency tree using a service to store HTML snippets

We can see that we have now abstracted away the use of our global variable. We still have a global area to store these HTML snippets, i.e. the Snippet Storage area, but we are now programming against an interface – that the `SnippetService` provides – and not against a concrete implementation. Our application is now guarded against any changes to the internal storage of these HTML snippets. As an example, we may decide to change our implementation from using HTML files, to storing HTML snippets in a database. In this case, only the internals of the `SnippetService` would need to be modified, and our code could carry on without needing to be changed.

Obviously, we will need some sort of key to allow us to store more than one snippet, but should the `SnippetService` be responsible for defining this key or not? Think Single Responsibility. Is the `SnippetService` really responsible for managing the keys that relate to the snippets? In other words, does it need to add or remove these keys? Not really. A smaller enum class would prove quite useful here, and favors numerous smaller interfaces over one general-purpose interface – think Interface Segregation.

With these things in mind, we can define the interface for our `SnippetService` as follows:

```
enum SnippetKey {
    CONTACT_ITEM_SNIPPET,
    OTHER_SNIPPET,
}

interface ISnippetService {
    storeSnippet(key: SnippetKey, value: string): void;
    retrieveSnippet(key: SnippetKey): string;
}
```

Firstly, we define an enum named `SnippetKey` to store all the keys to be used by the `SnippetService`. Secondly, we define the interface for the actual `SnippetService`, named `ISnippetService`, which has two functions. The first function will be a method to store a snippet, and is named `storeSnippet`. This function has two arguments, the first being a `SnippetKey` enum value, and the second argument is, of course, the HTML snippet itself. Similarly, the second function, named `retrieveSnippet` uses a single `SnippetKey` argument to retrieve the HTML snippet.

Now that we have defined an interface, we can create the structure of our `SnippetService` class:

```
class SnippetService implements ISnippetService {
  public storeSnippet(key: SnippetKey, value: string) {
  }
  public retrieveSnippet(key: SnippetKey) {
    return "";
  }
}
```

Here, we have a class named `SnippetService` that implements our `ISnippetService` interface. We have created the two methods defined in the interface, but have not yet provided an implementation. We will use this opportunity to follow TDD principles and write a failing unit test before writing the code that makes the tests pass. Our unit test is as follows:

```
describe("/tscode/tests/services/SnippetServiceTests.ts", () => {
  it("should store a snippet", () => {
    var snippetService = new SnippetService();
    snippetService.storeSnippet(
      SnippetKey.CONTACT_ITEM_SNIPPET, "contact_snippet");
    expect(
      snippetService.retrieveSnippet(
        SnippetKey.CONTACT_ITEM_SNIPPET)
    ).toBe("contact_snippet");
  });
});
```

In this test, we simply create an instance of the `SnippetService`, store a snippet with the key of `SnippetKey.CONTACT_ITEM_SNIPPET`, and then call `retrieveSnippet` with the same key, verifying the string value returned. Bear in mind that this is a simulated test, and in the real application, the `storeSnippet` call will occur during application initialization, and the `retrieveSnippet` call will occur at a later stage.

Let's now flesh out the `SnippetService` so that the tests pass:

```
class SnippetService implements ISnippetService {
  private snippetArray: string[] = new Array();
  public storeSnippet(key: SnippetKey, value: string) {
    this.snippetArray[key] = value;
  }
  public retrieveSnippet(key: SnippetKey) {
    if (!this.snippetArray[key]) {
      throw new Error(
        "SnippetService no snippet with key : " + key);
    }
    return this.snippetArray[key];
  }
}
```

Our `SnippetService` class now has an internal array of strings named `snippetArray`, marked as `private`, which will hold our HTML snippet values. Our `storeSnippet` and `retrieveSnippet` functions are now simply storing or retrieving values from this array. With this code in place, our test will now pass and our simple `SnippetService` is complete.

Dependency Resolution

Thus far, we have refactored our code to be dependent on an interface instead of a concrete object. This is all well and good, but begs the question: "how do we get hold of an interface?" – or more correctly – "how do we get hold of the concrete class that is currently implementing this interface?". This is the essential question that Dependency Injectors seek to answer.

There are a number of different ways in which a class can get hold of another class that implements an interface.

Service Location

If the class itself requests a concrete object based on an interface, then this process is called "**Service Location**". In other words, the class is using a registry or helper to locate the service it requires. You could also describe this technique as "dependency requesting". A central registry holds a lookup table with all registered classes against their respective interfaces. When the interface is requested, the Service locator simply looks up what class instance is stored against the interface in its table, and returns the object from its registry.

Dependency Injection

If the act of creating an instance of a class can be handed over to some sort of framework, then this framework can work out what interfaces a class needs, and "inject" these dependencies during class instantiation. This injection of dependencies is also called **assembly**. In this case, an assembler class or framework would need to be able to query an object to find out what interfaces it is dependent on. Unfortunately, we do not have this ability in JavaScript or TypeScript, as all interfaces are compiled away. So, we cannot use TypeScript interfaces by themselves to implement dependency injection. If we were to implement dependency injection in TypeScript or JavaScript, we would need some sort of naming convention to flag to the assembler framework that we need a concrete object to replace an interface.

Dependency Injection is also referred to as Inversion of Control – as we are handing over control of creation of our classes, and the resolution of their dependencies – to a third party. By the time we receive an instance of our class, all of the services or dependencies have been "magically" filled in.

Service Location versus Dependency Injection

The ideas around the Service Location pattern were first introduced by Martin Fowler around 2004, in a blog titled *Inversion of Control Containers and the Dependency Injection pattern* (<http://martinfowler.com/articles/injection.html>). However, in his book, *Dependency Injection in .NET*, Mark Seeman argues that the Service Location pattern is in fact an anti-pattern.

Mark's take on Martin's original ideas are that it is too easy to introduce runtime errors, or to misunderstand the usage of a particular class, when Service Location is used. This is because figuring out what services a class uses, means reading through the entire class. He argues that a better way of using Dependency Injection, is to list all dependencies in the constructor function of a class, and let the service locator resolve each dependency, during class construction. Most of Mark's examples seem to revolve around building and using APIs, where internals of a particular class cannot simply be read from the code, and using a class without knowing what services it depends on, can easily cause runtime errors.

While his ideas do certainly hold true, the solutions to this problem are all relevant to the .NET language – which has a key language feature that is unavailable in JavaScript – called Reflection. Reflection is the ability of a program – at runtime – to interrogate an object for information about itself, such as what properties it has, and what interfaces it implements or expects. Even though TypeScript provides the interface keyword, and does compile-time checking on these interfaces, all interfaces are compiled away in the resultant JavaScript.

This provides us with a serious problem. If a class is dependent on an interface, we cannot use this interface at runtime to look up the concrete implementation for the interface – because at runtime, this interface simply does not exist.

Angular uses a naming convention (a \$ prefix) to provide dependency injection capabilities. This has been rather successful, although there are caveats and some work-arounds when using minification routines. Angular 2.0 also solves this problem by providing a custom syntax to denote places where dependencies need to be injected. Other JavaScript frameworks – such as ExtJs – provide a mechanism to create objects by using a global creation routine, which then allows the framework to inject dependencies. This ExtJs technique, unfortunately, is not very compatible with the TypeScript language syntax (see *Chapter 5, Third Party Libraries* where we discuss ExtJs).

Also, if we are not using Angular, Angular 2.0, ExtJs, or any other framework, then Dependency Injection is just slightly out of reach in standard JavaScript. Service Location, on the other hand, can be accomplished, and combined with TypeScript interfaces, can bring us all of the benefits of dependency resolution and therefore, modular programming.

We can also make a compromise in order to incorporate the ideas that Mark suggests – and limit our Service Location to object constructors. When writing libraries that use Service Location, we would need to clearly document what dependencies a particular class has – and how they need to be registered. Even popular .NET Dependency injection frameworks such as StructureMap still allow for Service Location techniques – although they are being deprecated.

For the purposes of this book, then, let's explore how to write a simple Service Locator and use it in our code to build a more modular application, and leave the argument about pattern versus anti-pattern to those languages that have the features to implement Dependency Injection naturally.

A Service Locator

Let's get back to the crux of our problem: given an interface, how do we obtain a concrete implementation of a class that is currently implementing it?

In *Chapter 3, Interfaces, Classes and Generics*, we wrote a generic class named `InterfaceChecker` that did a runtime evaluation of a class, to check whether it implemented a specific set of methods and properties. The basic idea behind this `InterfaceChecker` was that if we provided a metadata class that listed the expected properties and methods of an interface, we could then interrogate a class at runtime against this metadata. If the class had all of the required properties and methods, then it was said to implement the interface.

So, we now have a mechanism – at runtime – to ensure that a class implements an interface: not a TypeScript interface, mind you, but a metadata-defined interface. If we extend this idea, and give each of our metadata interfaces a unique name, we have the concept of a "named interface". As long as these interface names are unique across our application, we now have a mechanism to query a class – at runtime – and see whether it implements a named interface.

If a class implements a named interface, we can then use a registry to store an instance of this class against its named interface. Any other code that needs an instance of a class that is implementing this named interface, simply has to query the registry, supply the interface name, and the registry will be able to return the class instance.

As long as we ensure that our TypeScript interfaces match the named interface definitions, we are all good to go.

Named interfaces

Back in *Chapter 3, Interfaces, Classes and Generics*, we wrote an interface named `IInterfaceChecker` that we could use as a standard template for our metadata. Let's update this interface and give it a required `className` property – so that we can use it as a named interface:

```
interface IInterfaceChecker {
    methodNames?: string[];
    propertyNames?: string[];
    className: string;
}
```

We still have the optional arrays of `methodNames` and `propertyNames`, but now every class that implements this interface will also require a `className` property.

So, given the following TypeScript interface:

```
interface IHasIdProperty {
    id: number;
}
```

Our named interface metadata class to match this TypeScript interface would look like this:

```
class IIHasIdProperty implements IInterfaceChecker {
    propertyNames: string[] = ["id"];
    className: string = "IIHasIdProperty";
}
```

This `IHasIdProperty` interface has a single property named `id`, which is of type `number`. We then create a class named `IHasIdProperty` to act as a named interface definition. This class implements our updated `IInterfaceChecker` interface and must, therefore, provide a `className` property. The `propertyNames` property has a single array entry named `id`, and will be used by our `InterfaceChecker` class to match against the `id` property of our TypeScript interface.

Note the naming convention of this class—it is the same name as the interface but adds an extra `I`. This double `I` convention will help us to tie the TypeScript interface named `IHasIdProperty` with its `IHasIdProperty` metadata named interface class.

We can now create a normal TypeScript class that implements the `IHasIdProperty` TypeScript interface as follows:

```
class PropertyOne implements IHasIdProperty {
  id = 1;
}
```

We now have all of the pieces in place to start building a Service Locator:

- A TypeScript interface named `IHasIdProperty`. This will provide compile-time type checking against a class implementing this interface.
- A named interface or metadata class called `IHasIdProperty`. This will provide runtime type checking against a class, and it also has a unique name.
- A class that implements the TypeScript interface `IHasIdProperty`. This class will pass the runtime type checks, and an instance of this class can be registered with our Service Locator.

Registering classes against named interfaces

With these metadata classes in place, we can now create a central repository to act as a Service Locator. This class has static functions for registering classes, as well as resolving interfaces:

```
class TypeScriptTinyIoC {
  static registeredClasses: any[] = new Array();
  public static register(
    targetObject: any,
    targetInterface: { new (): IInterfaceChecker; }): void {
  }

  public static resolve(
    targetInterface: { new (): IInterfaceChecker; }): any {
  }
}
```

```
    public static clearAll() {}  
}
```

This class, named `TypeScriptTinyIoC`, has a single static property named `registeredClasses`, which is an array of type `any`. This array is essentially our registry. As we do not know what type of class we are going to store in this array, the use of the `any` type in this instance is correct.

This class then provides two primary static functions, named `register` and `resolve`. The `register` function takes a `targetObject` as its first parameter, and then a class definition of a named interface—i.e. a class derived from `IInterfaceChecker`. Note the syntax of the `targetInterface` argument—it is the same as the generic syntax that we used in *Chapter 3, Interfaces, Classes and Generics*, to denote a class definition.

It is actually easier to understand these function signatures if we take a look at an example of their usage, so let's write a quick test:

```
it("should resolve instance of IProperty to PropertyOne", () => {  
    var propertyInstance = new PropertyOne();  
    TypeScriptTinyIoC.register(propertyInstance, IHasIdProperty);  
  
    var iProperty: IHasIdProperty =  
        TypeScriptTinyIoC.resolve(IHasIdProperty);  
    expect(iProperty.id).toBe(1);  
});
```

This test first creates an instance of the `PropertyOne` class, which implements the `IHasIdProperty` interface. This class is the one that we would like to register. The test then calls the `register` function of `TypeScriptTinyIoC` with two parameters. The first parameter is the class instance itself, and the second parameter is the class definition for the associated named interface—`IHasIdProperty`. We have seen this type of syntax before, when we discussed creating instances of classes using generics, but its signature is also available on nongeneric functions.

Without using the `targetInterface: { new (): IInterfaceChecker; }` signature, we would have to call this function as follows:

```
TypeScriptTinyIoC.register(propertyOneInstance,  
    new IHasIdProperty());
```

But with this signature in place, we can defer the creation of the `IHasIdProperty` named interface class to the `register` function—and drop the new syntax as follows:

```
TypeScriptTinyIoC.register(propertyOneInstance, IHasIdProperty);
```

Our test then calls the `resolve` function on `TypeScriptTinyIoC`, and again passes the class definition of our named interface as the lookup key. Finally, we check whether the class that is returned is in fact an instance of the `PropertyOne` class that we registered initially.

At this stage, our test will fail dramatically, so let's flesh out the `TypeScriptTinyIoC` class, starting with the `register` function:

```
public static register(  
    targetObject: any,  
    targetInterface: { new (): IInterfaceChecker; })  
{  
    var interfaceChecker = new InterfaceChecker();  
    var targetClassName = new targetInterface();  
    if (interfaceChecker.implementsInterface(  
        targetObject, targetInterface)) {  
        this.registeredClasses[targetObject.className]  
            = targetObject;  
    } else {  
        throw new Error(  
            "TypeScriptTinyIoC cannot register instance of "  
            + targetClassName.className);  
    }  
}
```

This `register` function firstly creates an instance of the `InterfaceChecker` class, and then creates an instance of the class definition passed in, through the `targetInterface` argument. This `targetInterface` is the named interface or metadata class. We then call the `implementsInterface` function of `interfaceChecker` to ensure that the `targetObject` implements the interface described by `targetInterface`. If it passes this check, we then add it to our internal array named `registeredClasses`, using the `className` property as a key.

Again, using our `InterfaceChecker` gives us runtime type checking – so that we can be sure that any class we are registering does in fact implement the correct named interface.

Now we can flesh out the `resolve` function as follows:

```
public static resolve(  
    targetInterface: { new (): IInterfaceChecker; })  
{  
    var targetClassName = new targetInterface();
```



```
    if (this.registeredClasses[targetClassName.className]) {
        return this.registeredClasses[targetClassName.className];
    } else {
        throw new Error(
            "TypeScriptTinyIoC cannot find instance of "
            + targetClassName.className);
    }
}
```

This `resolve` function only has one parameter—the definition of our named interface. Again, we are using the `new-able` syntax that we have seen previously. This function simply creates an instance of the `targetInterface` class, and then uses the `className` property as the key into the `registeredClasses` array. If an entry is found, we simply return it; otherwise, we throw an error.

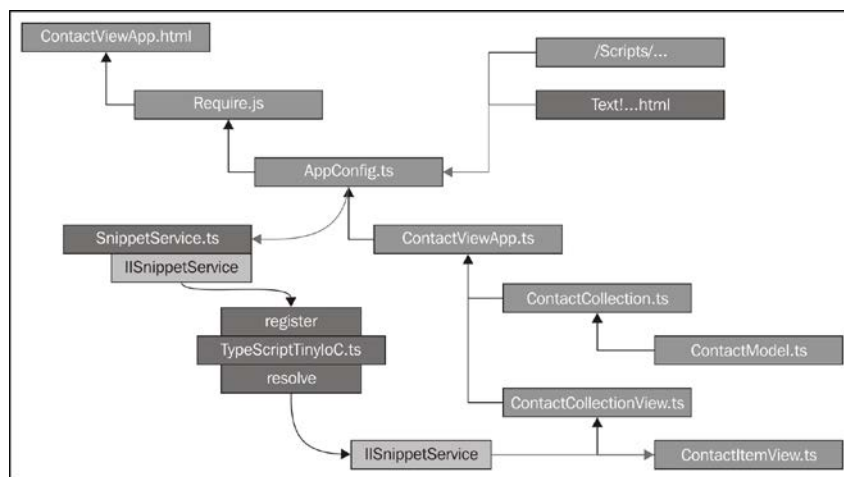
The final function on our `TypeScriptTinyIoC` class is the `clearAll` function, and it is used primarily in testing to clear out our registered classes array:

```
public static clearAll() {
    this.registeredClasses = new Array();
}
```

Our service locator is now complete.

Using the Service Locator

Let's now update our dependency tree to see how the `TypeScriptTinyIoC` service locator would be used:



Dependency diagram with a service locator pattern

Our `AppConfig.ts` code will now create an instance of the `SnippetService`, and register it with `TypeScriptTinyIoC` using a named interface – `IISnippetService`. Our `ContactItemView` constructor will then be updated to resolve an instance of the `IISnippetService` named interface from the registry. In this way, we are now programming to an interface – the `IISnippetService` interface. We use this named interface when we register our service with the service locator, and again when we resolve the service later on. Our `ContactItemView`, then, is asking the service locator to give us the current object that is implementing the `IISnippetService` interface.

To implement this change, we will firstly need a named interface to match the `ISnippetService` TypeScript interface. As a refresher, our `ISnippetService` was defined as follows:

```
interface ISnippetService {
    storeSnippet(key: SnippetKey, value: string): void;
    retrieveSnippet(key: SnippetKey): string;
}
```

Using our naming rules, our named interface definition would be called `IISnippetService` as follows:

```
class IISnippetService implements IInterfaceChecker {
    methodNames: string[] = ["storeSnippet", "retrieveSnippet"];
    className: string = "IISnippetService";
}
```

Note how the `methodNames` array contains two entries that match our TypeScript interface. By convention, we have also specified a `className` property, so that we can use this class as a named interface. Using the name of the class (`IISnippetService`) as the `className` property will also ensure a unique name, as TypeScript will not allow multiple class definitions with the same name.

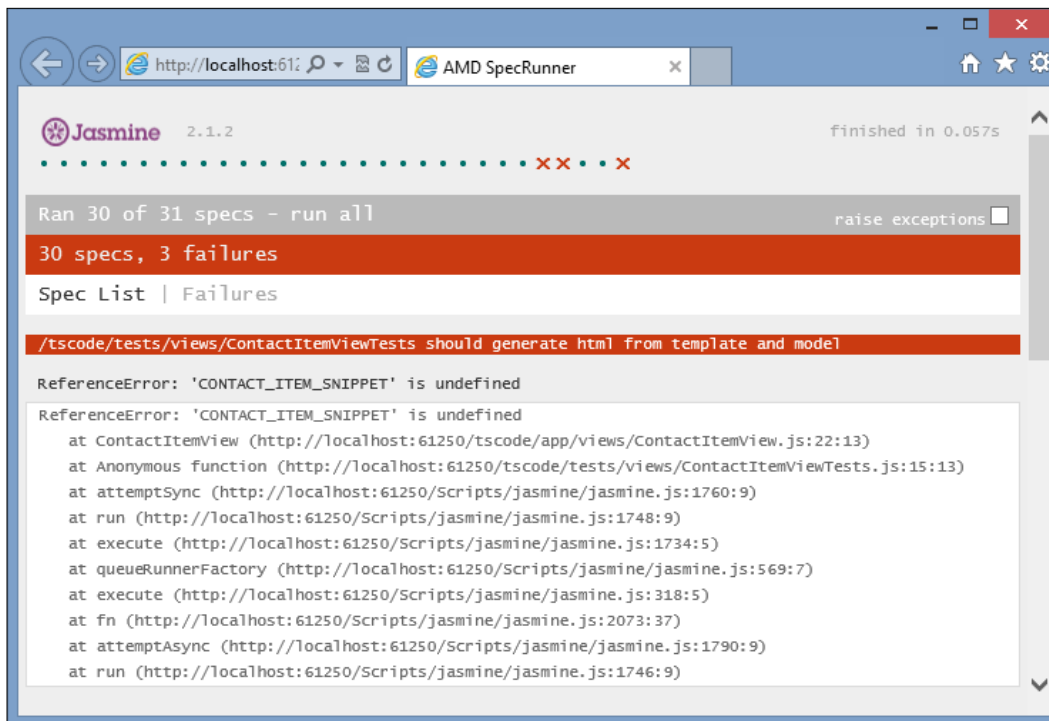
Let's now focus on our test suite. Remember that our `TestConfig.ts` file is almost identical to our `AppConfig.ts` file, but starts the Jasmine test suite instead of running our app. We will modify this `TestConfig.ts` file to include our `SnippetService` and `TypeScriptTinyIoC` as follows.

```
require.config(
    {
        // existing code
        paths: {
            // existing code
        }
    }
);
```

```
        'tinyioc': '/tscode/app/TypeScriptTinyIoC',
        'snippetservice': '/tscode/app/services/SnippetService'
    },
    shim: {
        // existing code
    }
}
);

require(
    ['jasmine-boot', 'tinyioc', 'snippetservice',
    'text!/tscode/app/views/ContactItemView.html'],
    (jb, tinyioc, snippetservice, contactItemSnippet) => {
        var snippetService = new SnippetService();
        snippetService.storeSnippet(
            SnippetKey.CONTACT_ITEM_SNIPPET,
            contactItemSnippet);
        TypeScriptTinyIoC.register(
            snippetService, IISnippetService);
        require(specs, () => {
            (<any>window).onload();
        });
    }
);
```

Firstly, we have included an entry for `tinyioc` and `snippetservice` in our `paths` property, to ensure that `Require` will load our files from the specified directory. We then update the call to the `require` function to include both the `tinyioc` and `snippetservice` in both of the arguments. Our anonymous function then creates a new instance of the `SnippetService` and stores the snippet that is loaded by `Text`, using the `CONTACT_ITEM_SNIPPET` key. We then register the instance of this `SnippetService` with `TypeScriptTinyIoC` using the named interface `IISnippetService`. If we run our test suite now, we should get a few failing tests:



Unit test failures

This failure is caused because the `ContactItemView` still references the `CONTACT_ITEM_SNIPPET` global variable. Let's now modify this view's constructor as follows:

```

constructor(options?: any) {
  var snippetService: ISnippetService =
    TypeScriptTinyIoC.resolve(IISnippetService);
  var contactItemSnippet = snippetService.retrieveSnippet(
    SnippetKey.CONTACT_ITEM_SNIPPET);

  this.className = "contact-item-view";
  this.events = <any>{ 'click': this.onClicked };
  this.template = _.template(contactItemSnippet);

  super(options);
}

```

The first line of the constructor calls the `TypeScriptTinyIoC.resolve` function with the definition of the named interface, `IISnippetService`. The result of this call is stored in the `snippetService` variable, which is strongly typed to the `ISnippetService` interface. This is the essence of the service locator pattern: we are programming to an interface (`ISnippetService`) and also locating this interface via our service locator. Once we have an instance of the class providing the interface, we can simply call `retrieveSnippet` with the required key to load our template.

Now that we have updated and fixed our tests, we will just need to modify our `AppConfig.ts` file in the same way that we modified the `TestConfig.ts` file.

Testability

Now that we are programming against a defined interface, we can start to test our code in different ways. In a test, we can now substitute the actual `SnippetService` for another service that throws an error when we call `retrieveSnippet`. For this test, let's create a class named `SnippetServiceRetrieveThrows` as follows:

```
class SnippetServiceRetrieveThrows implements ISnippetService {
    storeSnippet(key: SnippetKey, value: string) {}

    retrieveSnippet(key: SnippetKey) {
        throw new Error("Error in retrieveSnippet");
    }
}
```

This class can be registered against the `IISnippetService` named interface, as it correctly implements the TypeScript interface `ISnippetService`. The `retrieveSnippet` function, however, simply throws an error.

Our tests, then, can easily register this version of the service, and then create a `ContactItemView` class instance in order to see what happens, should the call to the `retrieveSnippet` function fail. Note that we have not modified our `ContactItemView` class in any way – we are simply registering a different class against the `IISnippetService` named interface. Our test, in this case, would be as follows:

```
beforeAll(() => {
    var errorService = new SnippetServiceRetrieveThrows();
    TypeScriptTinyIoC.register(errorService, IISnippetService);
});

it("should handle an error on constructor", () => {
    var contactModel = new cm.ContactModel(
```

```

        { Name: 'testName', EmailAddress: 'testEmailAddress' });

    var contactItemView = new ccv.ContactItemView(
        { model: contactModel });
    var html = contactItemView.render().$el.html();
    expect(html).toContain('error');

});

```

In this test, we are registering our throwing version of the `SnippetService` in our `beforeAll` function, and then testing the rendering capability of the `ContactItemView`. Running this test will cause an error to be thrown when the `ContactItemView` calls `retrieveSnippet`. To enable this test to pass, we need to update the `ContactItemView` to handle an error gracefully:

```

var contactItemSnippet = "";
var snippetService: ISnippetService =
    TypeScriptTinyIoC.resolve(IISnippetService);
try {
    contactItemSnippet = snippetService.retrieveSnippet(
        SnippetKey.CONTACT_ITEM_SNIPPET);
} catch (err) {
    contactItemSnippet =
        "There was an error loading CONTACT_ITEM_SNIPPET";
}

```

Here, we have simply surrounded the call to `retrieveSnippet` with a try catch block. If an error occurs, we are then modifying the snippet to be a standard error message. By putting a test like this in place, we are further solidifying our code to be able to handle various errors.

So what have we accomplished thus far? We have built a service to provide HTML snippets, and we have built a Service Locator that can register an instance of this service for use throughout our code. By registering different variations of this service during testing, we can also further bug-proof our code by simulating common errors, and testing our components under these circumstances.

The Domain Events Pattern

Most JavaScript frameworks have the concept of an event bus. An event bus is simply a method of publishing events to a global bus, so that other parts of your application that are subscribed to these events will receive a message, and be able to react to them. The use of an event-based architecture helps to decouple our applications, making them resilient to change and easier to test.

A Domain Event is an event that happens specific to our application domain. Something like "when an error occurs, log it to the console", or "when a menu button is clicked, change the sub-menu panel to reflect this option". A Domain Event can be raised anywhere in your code. Any class can register an event handler against this event, and will then be notified when this event is raised. There can be many event handlers for a single Domain Event.

Martin Fowler first blogged about the concept of a Domain Event in 2005 in a blog found at <http://martinfowler.com/eaaDev/DomainEvent.html>. Udi Dahan then showed how to implement a simple domain event pattern in C# in another blog found at <http://www.udidahan.com/2009/06/14/domain-events-salvation/>. Mike Hadlow also blogged about Separation of Concerns with Domain Events, and this blog can be found at <http://mikehadlow.blogspot.com.au/2010/09/separation-of-concerns-with-domain.html>.

Mike argues that a piece of code that raises an event should not be concerned with what happens after that – we should have separate handlers to handle these events – which are not coupled to anything actually raising the events.

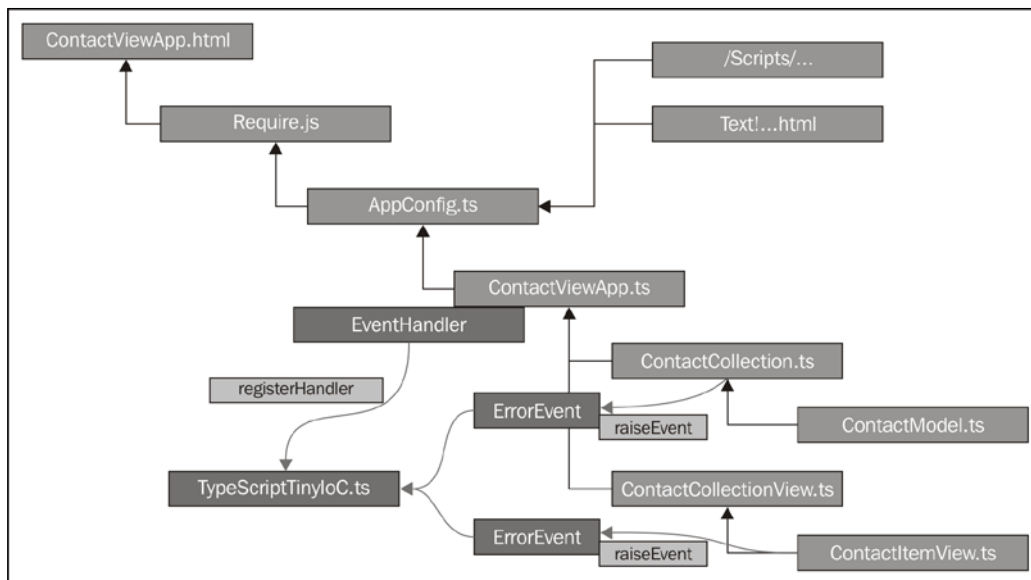
While there are a number of JavaScript libraries that handle events – Postal for example – most of these libraries send strings or simple JavaScript objects as the message packet. There is no way of ensuring that the sender of the message is filling in all of the properties that the handler of the message is expecting. In other words, these messages are not strongly typed – and could easily cause runtime errors – by trying to fit a "square peg" message into a "round hole" event handler.

In this section, we will build a strongly typed Domain Event message bus, and show how both sides – the event raiser and the event handler – can ensure that the event that is raised has all of the properties that are expected in the event handler. We will also show how to ensure that the event handlers are written correctly – and registered correctly – so that events are delivered in a strongly typed manner.

Problem space

Let's assume that we have the following business requirement: "If an error occurs, show the user an error message in a notification pop up. This pop up should show for two seconds and then fade away, allowing the user to continue working."

In our current application, there are a number of places where errors could occur – when loading JSON through the `ContactCollection`, for instance – or when rendering a `ContactItemView`. These errors could occur quite deep down in our class hierarchy. In order to achieve our stated requirements, we will need to handle these errors at the `ContactViewApp` level. Consider the following diagram:



Dependency tree with domain event handlers and event raisers.

Our `ContactViewApp` will register an event handler with `TypeScriptTinyIoC`, specifying which event type it is interested in. When an event of this type is raised by any one of our modules, our message bus will direct the message to the correct handler, or group of handlers. In the preceding diagram, the `ContactCollection` and the `ContactItemView` classes are shown to be raising an `ErrorEvent` via `TypeScriptTinyIoC`.

Message and Handler Interfaces

There are two key sets of information that we need in order to register and raise strongly typed messages. The first is an interface describing the message itself, which is paired with its named interface. The second is an interface describing the message handler function, again which is paired with its named interface. Our `TypeScript` interface gives us compile-time checking of messages and handlers, and our named interfaces (implementing `IInterfaceChecker`) give us runtime type checking of messages and handlers.

First up, the interfaces for our message are as follows:

```
interface IErrorEvent {
    Message: string;
    Description: string;
}

export class IErrorEvent implements IInterfaceChecker {
    propertyNames: string [] = ["Message", "Description"];
    className: string = "IErrorEvent";
}
```

We start with the TypeScript interface `IErrorEvent`. This interface has two properties, `Message` and `Description`, which are both strings. We then create our `IErrorEvent` class, which is an instance of our named interface – again with the `propertyNames` array matching our TypeScript interface property names. The `className` property is also set to be the name of the class, `IErrorEvent`, to ensure uniqueness.

The interfaces for our event handlers are then as follows:

```
interface IErrorEvent_Handler {
    handle_ErrorEvent(event: IErrorEvent);
}

export class IErrorEvent_Handler implements IInterfaceChecker {
    methodNames: string[] = ["handle_ErrorEvent"];
    className: string = "IErrorEvent_Handler";
}
```

The TypeScript interface `IErrorEvent_Handler` contains a single method, named `handle_ErrorEvent`. This handler method has a single parameter, `event`, which is again strongly typed to be our event interface, `IErrorEvent`. We then construct a named interface called `IErrorEvent_Handler`, and match the TypeScript interface through the `methodNames` array. Again, we provide a unique `className` property for this named interface.

With these two interfaces and named interfaces in place, we can now create the actual `ErrorEvent` class as follows:

```
export class ErrorEvent implements IErrorEvent {
    Message: string;
    Description: string;
    constructor(message: string, description: string) {
        this.Message = message;
        this.Description = description;
    }
}
```

The class definition for `ErrorEvent` implements the `IErrorEvent` interface, thereby making it compatible with our event handler. Note the `constructor` of this class. We are forcing users of this class to provide both a `message` and `description` parameter in the constructor – thereby using TypeScript compile-time checking to ensure that we construct this class correctly, no matter where it is used.

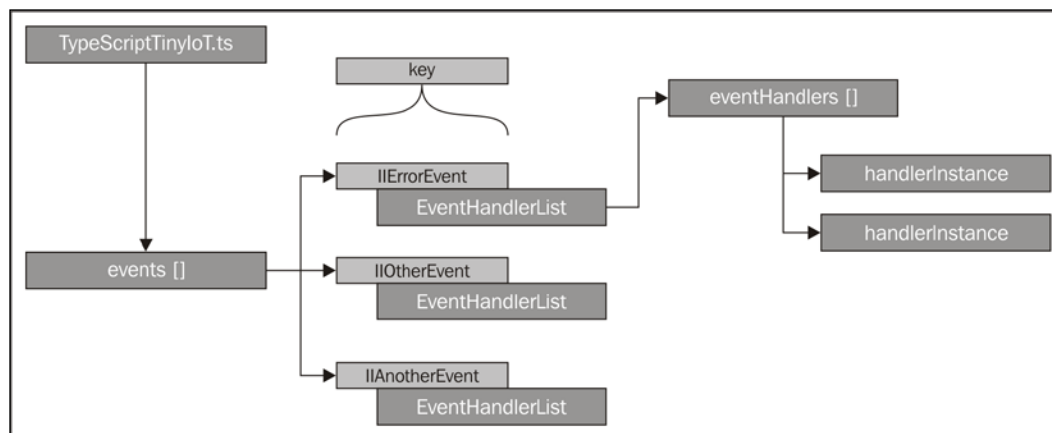
We can then create a class that implements the `IErrorEvent_Handler` interface, which will receive the event itself. As a quick example, consider the following class:

```
class EventHandlerTests_ErrorHandler
  implements IErrorEvent_Handler {
  handle_ErrorEvent(event: IErrorEvent) {
  }
}
```

This class implements the `IErrorEvent_Handler` TypeScript interface, and therefore the compiler will force the class to define a `handle_ErrorEvent` function with the correct signature, in order to receive messages.

Multiple Event Handlers

To be able to register multiple events, and have multiple event handlers per event, we will need an array of events, each of which will, in turn, hold an array of handlers as follows:



Class structure for registering multiple event handlers per event.

Our `TypeScriptTinyIoC` class will have an array called `events`, which uses the name of the event as its key. This name will be drawn from our named interface for the event – again because TypeScript interfaces are compiled away. To help with managing multiple event handlers per event, we will create a new class called `EventHandlerList` that will facilitate the registration of multiple event handlers. An instance of this `EventHandlerList` class will be stored in our `events` array for each named event that we have registered.

Let's start with this list of event handlers, and implement our `EventHandlerList` class. At this stage, all we need is an internal array to store handlers, named `eventHandlers`, along with a `registerHandler` function as follows:

```
class EventHandlerList {
    eventHandlers: any[] = new Array();
    registerHandler(handler: any,
        interfaceType: { new (): IInterfaceChecker }) {
    }
}
```

The `registerHandler` function is again using the `{ new(): IInterfaceChecker }` syntax for the `interfaceType` argument, thereby allowing us to use a type name for this function call. A quick unit test is as follows:

```
import iee = require("../app/events/ErrorEvent");

class EventHandlerTests_ErrorHandler
    implements iee.IErrorEvent_Handler {
    handle_ErrorEvent(event: iee.IErrorEvent) {
    }
}

describe("/tests//EventHandlerTests.ts", () => {

    var testHandler: EventHandlerTests_ErrorHandler;
    beforeEach(() => {
        testHandler = new EventHandlerTests_ErrorHandler();
    });

    it("should register an event Handler", () => {
        var eventHandlerList = new EventHandlerList();
        eventHandlerList.registerHandler(testHandler,
            iee.IErrorEvent_Handler);

        expect(eventHandlerList.eventHandlers.length).toBe(1);
    });
});
```

We start this test with an `import` statement for our event classes, and then a class named `EventHandlerTests_ErrorHandler`. This class will be used as a registered event handler just for this test suite. The class implements the `iee.IErrorEvent_Handler` and, as such, will generate a compile error if we do not have a `handle_ErrorEvent` function that accepts an `IErrorEvent` as its only parameter. Just by using TypeScript interfaces, we have already ensured that this class has the correct function name and function signature to accept `ErrorEvent` messages.

Our test then starts by declaring a variable named `testHandler` to store an instance of our `EventHandlerTests_ErrorHandler` class. The `beforeEach` function will create this instance, and assign it to our `testHandler` variable. The test itself then creates an instance of the `EventHandlerList` class, calls the `registerHandler`, and then expects the length of the internal `eventHandlers` property to be the value of one.

Note again the syntax of the call to `registerHandler`. We are passing in our `testHandler` instance as the first argument, and then specifying the named interface `IErrrorEvent_Handler` class type. As we saw with the service locator pattern, we are again using the same class name syntax for our named interface, instead of having to call `new()`.

Let's now fill in the code to make the test pass:

```
class EventHandlerList {
    eventHandlers: any[] = new Array();
    registerHandler(handler: any,
        interfaceType: { new (): IInterfaceChecker }) {

        var interfaceChecker = new InterfaceChecker();
        if (interfaceChecker.implementsInterface(
            handler, interfaceType)) {
            this.eventHandlers.push(handler);
        } else {
            var interfaceExpected = new interfaceType();
            throw new Error(
                "EventHandlerList cannot register handler of "
                + interfaceExpected.className);
        }
    }
}
```

Our `registerHandler` function firstly creates an instance of the `InterfaceChecker` class, and then calls `implementsInterface` to make sure, at runtime, that the handler object that is passed in does indeed have all of the method names defined by our named interface. If the `implementsInterface` function returns `true`, we can simply push this handler onto our internal array.

If the handler does not implement the named interface, we throw an error. For completeness, this error contains the `className` property of the named interface, so we first have to new up an instance of this named interface class, before we can extract the `className` property.

Let's now create a test that will deliberately fail our `implementsInterface` check and ensure that an error is in fact thrown:

```
class No_ErrorHandler {
}

it("should throw an error with the correct className", () => {
  var eventHandlerList = new EventHandlerList();
  expect(() => {
    eventHandlerList.registerHandler(new No_ErrorHandler(),
      iee.IIErrorEvent_Handler);
  }).toThrow(new Error(
    "EventHandlerList cannot register handler of
      IIErrorEvent_Handler"
  ));
});
```

We start with the class definition of the `No_ErrorHandler` class that obviously does not implement our named interface. Our test then sets up the `EventHandlerList` class, and calls the `registerHandler` function, using a new instance of the `No_ErrorHandler` class, and our `IIErrorEvent_Handler` named interface. We are then expecting a specific error message— one that should include the name of our named interface, `IIErrorEvent_Handler`.

Firing an event

We can now turn our attention to raising an event. To do this, we will need to know what the actual function name of the event handler is. We will make a slight change to our `EventHandlerList`, and pass in the event name to the constructor as follows:

```
class EventHandlerList {
  handleEventMethod: string;
  constructor(handleEventMethodName: string) {
    this.handleEventMethod = handleEventMethodName;
  }
}
```

```

    }

    raiseEvent(event: any) {
    }
}

```

Our constructor is now expecting a `handleEventMethodName` as a required parameter, and we are storing this in a property named `handleEventMethod`. Remember that all of the handlers that are registered with an instance of this class are responding to the same event – and as such will all have the same method name – enforced by the TypeScript compiler. We have also defined a `raiseEvent` function, and since we do not know what event this class will be handling, the event is of type `any`.

Now, we can create a unit test that will fail, as the `raiseEvent` function is not actually doing anything as yet. Before we do this, let's update our test handler class, `EventHandlerTests_ErrorHandler`, to store the last event fired in a property that we can access later:

```

class EventHandlerTests_ErrorHandler
  implements iee.IErrorEvent_Handler {
  LastEventFired: iee.IErrorEvent;
  handle_ErrorEvent(event: iee.IErrorEvent) {
    this.LastEventFired = event;
  }
}

```

We have updated this class definition with a property named `LastEventFired`, and set this property inside the `handle_ErrorEvent` function. With this change in place, when an event is fired, we can interrogate the `LastEventFired` property to see what event was fired last. Let's now write a test that calls the `raiseEvent` method:

```

it("should fire an event", () => {
  var eventHandlerList = new
    EventHandlerList('handle_ErrorEvent');
  eventHandlerList.registerHandler(testHandler,
    iee.IErrorEvent_Handler);
  eventHandlerList.raiseEvent(
    new iee.ErrorEvent("test", "test"));
  expect(testHandler.LastEventFired.Message).toBe("test");
});

```

We start with a variable named `eventHandlerList` that holds an instance of our `EventHandlerList` class, and pass in the name of the function to be called via the constructor. We then call `registerHandler` with this `testHandler` instance. Now, we can call the `raiseEvent` function, passing in a new `ErrorEvent`. As the constructor of our `ErrorEvent` class requires two parameters, we have just passed in "test" for each of these arguments. Finally, we are expecting that the `LastEventFired` property of our event handler to be set correctly. Running our test at this stage will fail, so let's implement the `raiseEvent` method on our `EventHandlerList` class as follows:

```
raiseEvent(event: any) {
    var i, len = 0;
    for (i = 0, len = this.eventHandlers.length; i < len; i++) {
        var handler = this.eventHandlers[i];
        handler[this.handleEventMethod](event);
    }
}
```

The implementation of this `raiseEvent` function is relatively simple. We just iterate through our `eventHandlers` array, and then get a reference to each of the event handlers using an index. The line to note here is how we execute the handler function: `handler[this.handleEventMethod](event)`. This takes advantage of JavaScript's ability to calling a function using a string value that matches the function's name. In our tests, this would be equivalent to `handler['handle_ErrorEvent'](event)`, which in JavaScript is equivalent to `handler.handle_ErrorEvent(event)` – an actual call to the handler function. With this JavaScript magic in place, our events are being fired, and our unit tests run through correctly.

Registering an Event handler for an Event

Now that we have a working, tested class to manage multiple event handlers responding to a specific event, we can turn our attention back to the `TypeScriptTinyIoC` class.

As we did for our Service Locator pattern, we will need to register an instance of an object to handle a specific event. The method signature for registering our event handler will look like this:

```
public static registerHandler(
    handler: any,
    handlerInterface: { new (): IInterfaceChecker },
    eventInterface: { new (): IInterfaceChecker }) {
}
```

This `registerHandler` function takes three arguments. The first is the instance of the object implementing the handler. The second argument is the named interface class for our handler – so that we can check this class at runtime to ensure that it implements the handler interface. The third argument is the named interface for the event itself. This `register` function is also what binds an event to its handler.

Before we put together a unit test, we will need another static function to raise an event:

```
static raiseEvent(event: any,
  eventInterface: { new (): IInterfaceChecker }) {
}
```

This `raiseEvent` function on the `TypeScriptTinyIoC` class will call the `raiseEvent` function on the `EventHandlerList` class instance for this event. We will also do an `interfaceChecker` test here, in order to ensure that the event being raised matches our named interface class for the event – before we actually raise the event.

Now for our unit test:

```
it("should register an event handler with
TypeScriptTinyIoC and fire an event", () => {
  TypeScriptTinyIoC.registerHandler(testHandler,
    iee.IIErrorEvent_Handler, iee.IIErrorEvent);
  TypeScriptTinyIoC.raiseEvent(
    new iee.ErrorEvent("test", "test"),
    iee.IIErrorEvent);
  expect(testHandler.LastEventFired.Message).toBe("test");
});
```

This test is very similar to the test that we wrote for our `EventHandlerList` class, except we are calling the `registerHandler` and `raiseEvent` methods on the `TypeScriptTinyIoC` class, instead of a specific `EventHandlerList`. With this failing test in place, we can now fill out the `registerHandler` and `raiseEvent` functions as follows:

```
static events: EventHandlerList[] = new Array<EventHandlerList>();
public static registerHandler(
  handler: any,
  handlerInterface: { new (): IInterfaceChecker },
  eventInterface: { new (): IInterfaceChecker }) {

  var eventInterfaceInstance = new eventInterface();
  var handlerInterfaceInstance = new handlerInterface();

  var handlerList =
```



```
        this.events[eventInterfaceInstance.className];
    if (handlerList) {
        handlerList.registerHandler(handler, handlerInterface);
    } else {
        handlerList = new EventHandlerList(
            handlerInterfaceInstance.methodNames[0]);
        handlerList.registerHandler(handler, handlerInterface);
        this.events[eventInterfaceInstance.className] =
            handlerList;
    }
}
```

Firstly, we have added a static property called `events`, which is an array of `EventHandlerList` instances. We will add to this array using the `className` of our named event interface as a key. Our `registerHandler` function firstly creates instances of both named interface classes that are passed in via the `handlerInterface` and `eventInterface` arguments. We are then checking to see whether our internal array already has an `EventHandlerList` instance for this event, keyed via the `className` property of our named event interface. If we have an entry already, we can simply call the `registerHandler` function on the existing `EventHandlerList` instance. If this event has not been registered, we simply create a new instance of an `EventHandlerList` class, call `registerHandler`, and then add this entry to our internal array.

Note how we figured out what the actual name of the event handler function call is. We are simply using the first method name found in our method names array: `handlerInterfaceInstance.methodNames[0]`, which will return a string. In our samples, this would return the `'handle_ErrorEvent'` string, which is the method name that we will need to invoke when invoking handler functions for an event.

Next, we can focus on the `raiseEvent` function:

```
static raiseEvent(event: any,
    eventInterface: { new (): IInterfaceChecker }) {

    var eventChecker = new InterfaceChecker();
    if (eventChecker.implementsInterface(event, eventInterface)) {
        var eventInterfaceInstance = new eventInterface();
        var handlerList =
            this.events[eventInterfaceInstance.className];
        if (handlerList) {
            handlerList.raiseEvent(event);
        }
    }
}
```

This function first creates an instance of an `InterfaceChecker` class, and then ensures that the event being raised conforms to the named interface that we provide as the second parameter. Again, this is a runtime type check to ensure that the event we are attempting to raise is in fact of the correct type. If the event is valid, we fetch the instance of the `EventHandlerList` class that is registered for this event, and then call its `raiseEvent` function.

Our strongly typed Domain Event mechanism is now complete. We are using both compile-time TypeScript interface checking, and runtime type checking in two ways. Firstly, when registering a handler, we do an interface check, and then when we fire an event, we do another interface check. This means that both sides—registering and firing—of events are strongly typed, both at compile time and also at runtime.

Displaying error notifications

Now that we have our `TypeScriptTinyIoC` event mechanism in place, we can focus on solving the business problem of showing error notifications when errors occur. `Notify` is a jQuery plugin that suits our needs perfectly (<http://notifyjs.com/>). We could install the JavaScript library from NuGet (Install the `jquery.notify` package), but the default version of this package relies on another package named `Bootstrap` for its styling. `Notify`, however, also provides an option on their website to download a custom `notify.js` script that has all of these styles built-in to the library. We will use this custom version, as our project is not using the `Bootstrap` package.

The definition file for `Notify` can be downloaded from `DefinitelyTyped` (<https://github.com/borisyanov/DefinitelyTyped/tree/master/notify>). At the time of writing, however, there seems to be two versions of the `Notify` library, one named `Notify` and the other named `Notify.js`. Use the `Notify` version as it seems to be more up to date.

To simulate an error, let's tag onto the `ContactItemView` `onClicked` function, where we are currently executing a flip, and raise a dummy error whenever someone clicks on one of our contact links:

```
onClicked() {
  this.$el.flip({
    direction: 'tb',
    speed : 200
  });
  var errorEvent = new iee.ErrorEvent(
    "Dummy error message", this.model.Name);
  TypeScriptTinyIoC.raiseEvent(errorEvent, iee.IIErrorEvent);
}
```

After our call to `flip`, we are simply creating an instance of the `ErrorEvent` class, with its two required parameters. We then call the `raiseEvent` function on `TypeScriptTinyIoC` with this `errorEvent` instance, and the named interface for the type of event that we are raising. It's as simple as that.

Now, we can modify our `ContactViewApp` to register a handler for this event as follows:

```
import iee = require("tscode/app/events/ErrorEvent");

export class ContactViewApp implements iee.IErrorEvent_Handler {
  constructor() {
    TypeScriptTinyIoC.registerHandler(this,
      iee.IErrorEvent_Handler, iee.IErrorEvent);
  }
  run() {

  }

  contactCollectionLoaded(model, response, options) {

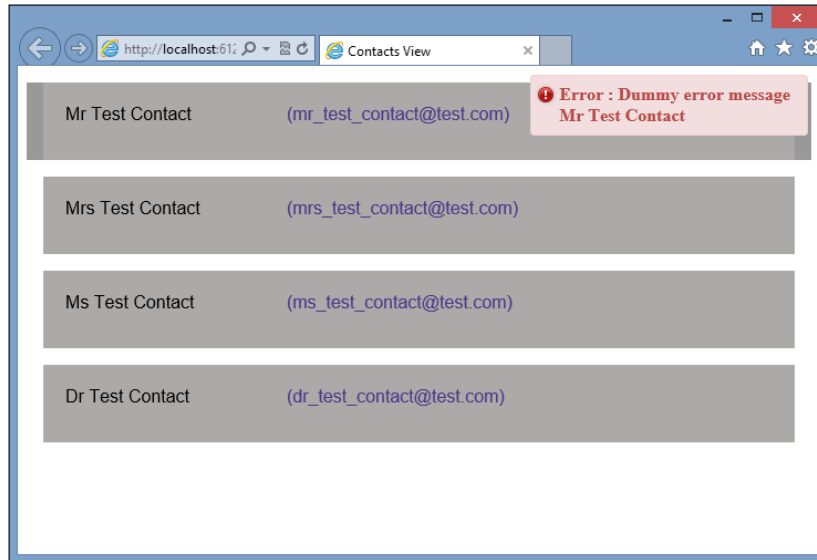
  }
  contactCollectionError(model, response, options) {

  }
  handle_ErrorEvent(event: iee.IErrorEvent) {
    $.notify("Error : " + event.Message
      + "\n" + event.Description);
  }
}
```

Here, we have made a few changes to our `ContactViewApp` class. Firstly, we implement the `IErrorEvent_Handler` TypeScript interface, which will force us to include the `handle_ErrorEvent` function within our class. We have also defined a constructor, and within this, we are registering the class instance as a handler using our two named interfaces: `IErrorEvent_Handler`, and `IErrorEvent`.

Within the `handle_ErrorEvent` function, we are calling `$.notify`—the Notify jQuery plugin. Note that the type of the event argument passed into the `handle_ErrorEvent` function, is of type `IErrorEvent`. This means that we can safely use any properties or methods of the `IErrorEvent` interface within our event handler function, as we have already ensured, during event raising, that this event implements the interface correctly.

Our call to Notify is just using a message that is built up from our `ErrorEvent`. The following screenshot shows the results of this Notify call:



Screenshot of application showing an error notification



The implementation of this Service Locator pattern and the strongly typed Domain Events pattern that we have worked through in this chapter are available on the GitHub project *typescript-tiny-ioc* (<https://github.com/blorkfish/typescript-tiny-ioc>). This project has further code samples as well as a full suite of unit tests for both AMD and normal JavaScript usage.

Summary

In this chapter, we had a look at object-oriented programming, beginning with the SOLID Design principles. We then reviewed the application that we had built in *Chapter 7, Modularization*, with regards to these principles. We discussed various methods of Dependency Injection, and then built a mechanism that is based on our *InterfaceChecker* from *Chapter 3, Interfaces, Classes and Generics*, to obtain an instance of a named interface. We used this principle to build a Service Locator and then extended this principle to build a strongly typed event bus for the Domain Event pattern. Finally, we incorporated Notify into our application for simple notifications in response to these error events. In our next and final chapter, we will put all of the principles we have learned so far into practice, and build an application from the ground up.

9

Let's Get Our Hands Dirty

In this chapter, we will look at building a TypeScript single-page web application from the ground up. We will start with a discussion on what the site should look like, how we want our page transitions to flow, and then move on to explore the capabilities of the Bootstrap framework, and discuss a pure HTML version of our site. Our focus will then switch to the data structures that we will need for our application, and what Backbone models and collections we need to represent this data. Along the way, we will write a set of unit and integration tests for these models and collections.

Once we have data to work with, we will then use the **Marionette** framework to build views in order to render our application to the DOM. We will then show how to break up our pure HTML version of the site into smaller portions of HTML snippets, and then integrate these snippets with our Marionette views. Finally, we will tie the application together using events, and explore the **State and Mediator** Design Pattern to help us manage complex page transitions and DOM elements.

Marionette

Marionette is an extension of the Backbone library, and introduces a number of enhancements to the framework, in order to reduce boilerplate Backbone code, and make working with DOM elements and HTML fragments easier. Marionette also introduces the concept of layouts and regions to help with managing logical portions of HTML within a large web page. A Marionette layout is a type of controller that manages several regions, and a Marionette region is an object that manages a particular HTML portion of our page. As an example, we could have a region for the header panel, one for a side-bar panel, and another for a footer area. This allows us to break up our application into logical areas, and then tie them together through messaging.

Bootstrap

We will also be using Bootstrap to help with our page layout. Bootstrap is a popular mobile-first framework for rendering HTML elements across a number of different platforms. Bootstrap styling and customization is a topic big enough for its own book, so we won't be exploring the ins and outs of the various Bootstrap options. If you are keen on learning more, then be sure to read the excellent book by David Cochran and Ian Whitley called *Bootstrap Site Blueprints, Packt Publishing* (<https://www.packtpub.com/web-development/bootstrap-site-blueprints>).

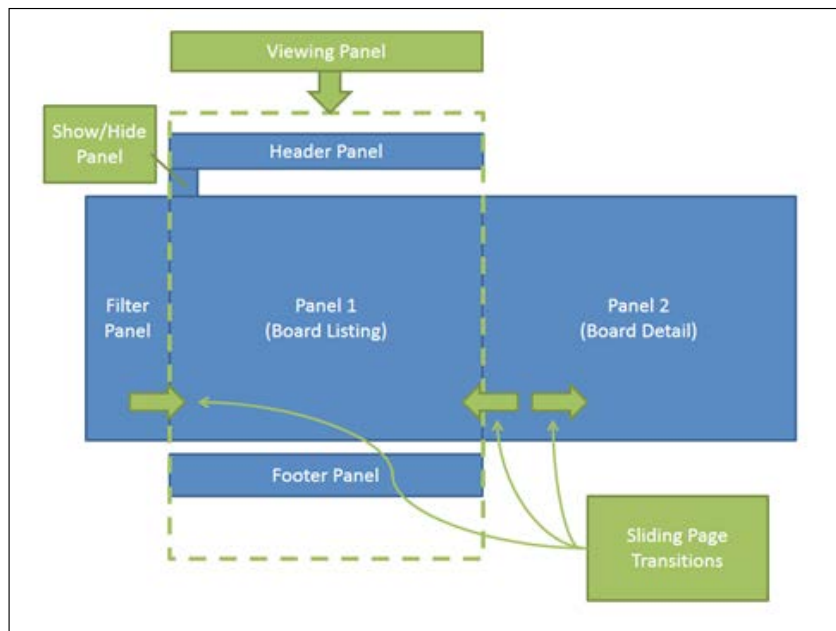
Board Sales

Our application will be a rather simple one, called Board Sales, and will list a range of windsurfing boards on the main page, using a summary view, or board list view. Clicking on any one of these boards will transition the page to show detailed information on the selected board. On the left-hand side of the screen, there will be a simple panel to allow the user to filter the main board list via manufacturer, or board type.

Modern windsurfing boards come in a range of sizes, and are measured by volume. Smaller volume boards are generally used for wave sailing, and larger volume boards are used for racing, or slalom. Those in-between can be categorized as freestyle boards, and are used for performing acrobatic tricks on flat water. Another important element of any board is the range of sails that the board is designed for. In very strong winds, smaller sails are used to allow the windsurfer to control the power generated by the wind, and in lighter winds, larger sails are used to generate more power. Our summary view will include a quick reference to the volume measurements for each board, and our detail view will show all the various board measurements and a compatible list of sail ranges.

Page layout

With this application, we will use the power of JavaScript to provide a left-to-right panel-style page layout. We will use some Bootstrap transitions to slide panels in from the left, or from the right, in order to provide the user with a slightly different browsing experience. Let's take a look at what this will look like conceptually:



A conceptual view of page transitions for Board Sales

The **viewing panel** will be our main page, with a **header panel**, a **board listing panel**, and a **footer panel**. Hidden from view on the left-hand side will be the **filter panel**, with a button on the top-left of the main panel to show or hide this filter panel. The filter panel will slide in from the left when needed, and slide back to the left when hidden. Similarly, the **board detail panel** will slide in from the right when a board is clicked, and will slide back to the right when the back button is clicked, revealing the board listing panel.

When the site is viewed on a desktop device, the filter panel on the left will be shown by default, but when the site is viewed on a tablet device – with a smaller screen – then the filter panel will be hidden by default, in order to save on screen real estate.

Installing Bootstrap

Bootstrap is a collection of CSS styles and JavaScript functions that aid in building responsive websites rather simply and easily. The responsive nature of Bootstrap means that pages will resize elements automatically, to allow rendering on the smaller screen sizes of mobile phones, as well as larger screens used on tablets and desktops. By using Bootstrap, we gain the additional benefit of being able to target mobile users and desktop users with very little change to our HTML or CSS style sheets.

Bootstrap can be installed with a NuGet package, along with the corresponding TypeScript definitions as follows:

```
Install-package bootstrap
```

```
Install-package bootstrap.TypeScript.DefinitelyTyped
```

Once Bootstrap has been installed, we can start building a sample web page that is purely written in HTML using Bootstrap. Building a demo page in this way helps us to figure out what Bootstrap elements we will use, and allows us to modify our CSS styles and structure our HTML correctly, before we start to build our application. This is where the Brackets editor really comes into its own. By using the live preview functionality of the editor, we can edit our HTML and CSS in one IDE, and have instant visual feedback in the preview pane. Working on sample HTML in this way is both a rewarding and fun experience, not to mention a massive time-saver.

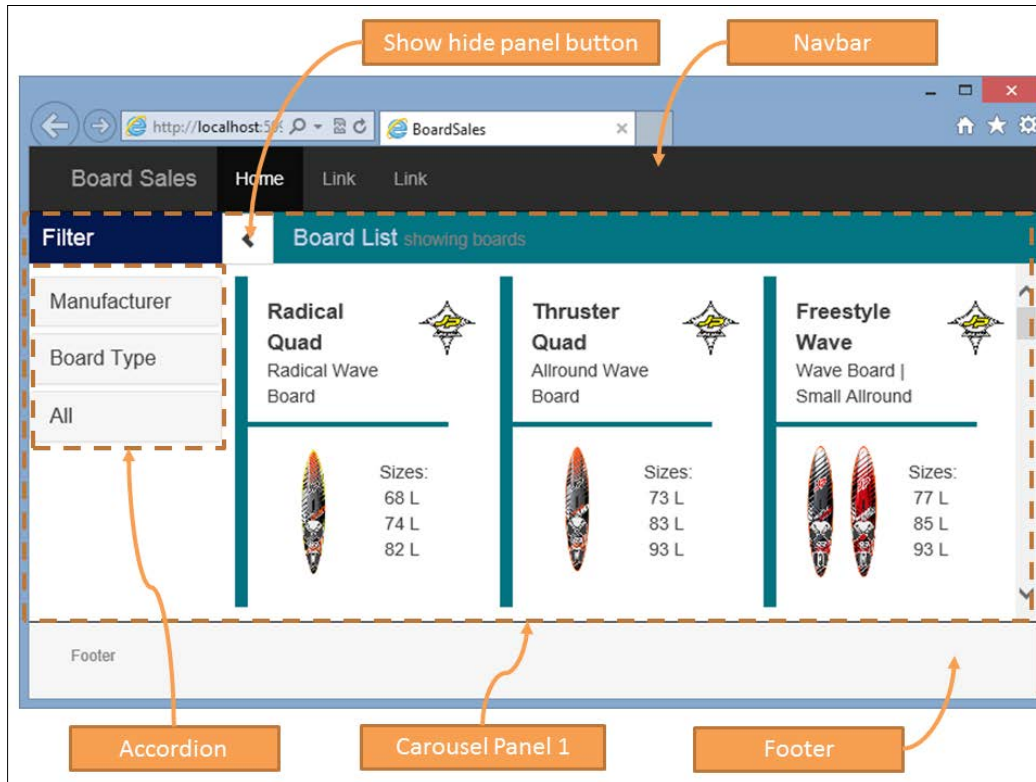
Using Bootstrap

Our page will use a couple of Bootstrap elements for the main page regions, as follows:

1. A **Navbar** component to render the header panel.
2. A **Footer** component to render the footer panel.
3. A **Carousel** component to slide from the board list view to the board detail view.
4. An **Accordion** component to render the filtering options in the left-hand side panel.
5. **Row** and **Column** components to control the HTML layout of boards in our board list view, as well as in the board detail view.
6. Table CSS elements to render tables.

In this chapter, we will not go into detail about how to build HTML pages with Bootstrap. We will instead start with a working version that you can find in the sample code under the directory `/tscode/tests/brackets/TestBootstrap.html`.

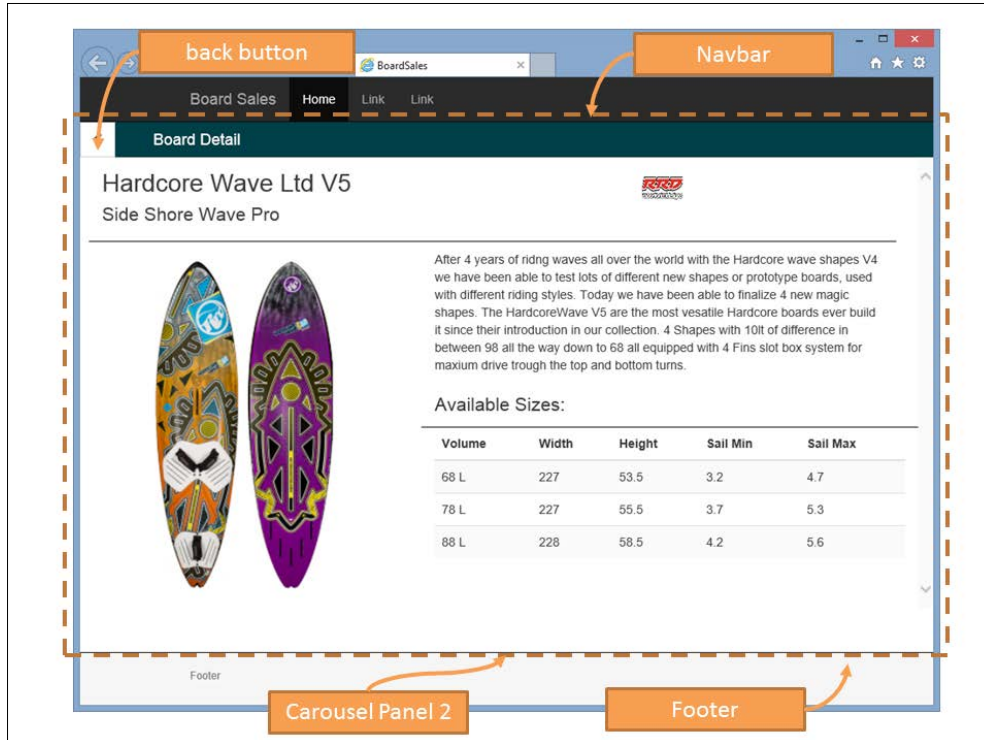
Our Bootstrap elements are as follows:



At the top of our page is the navbar element, which has been given a `navbar-inverse` style to render it with a black background. The **carousel panel 1** element is the first carousel panel, and contains the left-hand side filter panel, as well as the board list and the **show / hide panel** button. The **filter** options on the left-hand side panel use the Bootstrap accordion component. Finally, our footer is styled to be a "sticky footer", meaning that it will always show on the page.

When we click on any one of the boards in the board list, our carousel component will slide the carousel panel over to the left, and slide in the board detail view from the right.

Our board detail panel is as follows:



Again, we have the standard header and footer regions, but this time, we are viewing **carousel panel 2**. This panel has a back button on the top left-hand side, and shows the detailed information on the selected board.

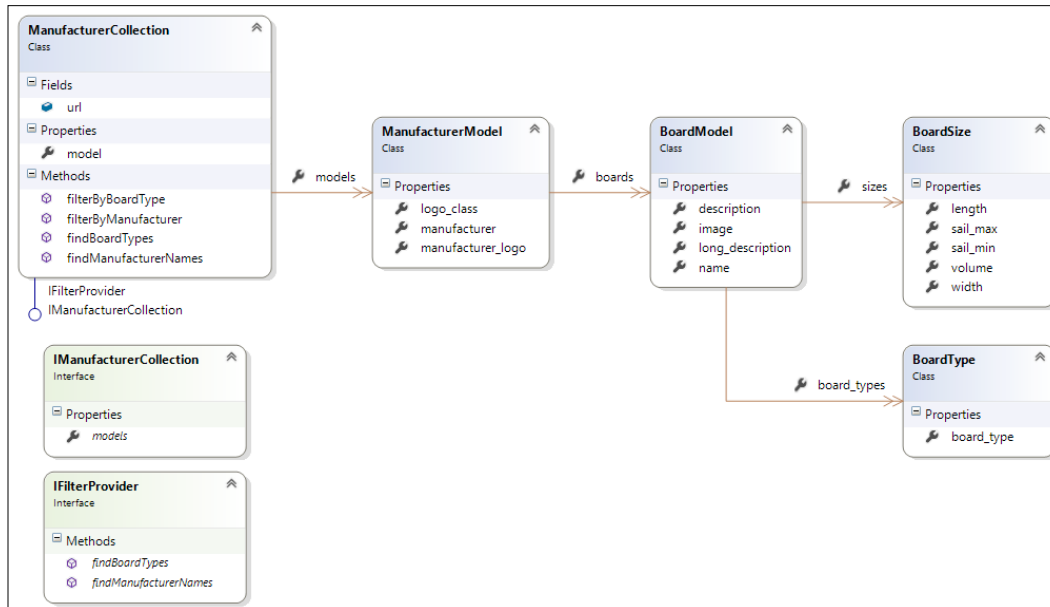
You will notice when you run this test page, that there are four links in the footer region named **next**, **prev**, **show**, and **hide**. These buttons are used to test the cycling of the carousel panels, and the show / hide functionality for the left hand-side panel.

Bootstrap is ideal for building quick mock-ups of a working version of the site. This version can easily be taken to customers, or to project meetings for demo purposes. Showing a customer a demo mockup of a site will give you invaluable feedback on the overall site's flow and design. Ideally, this sort of work should be done by a senior web designer, or someone with equal skill set – who specializes in CSS styling.

We will be reusing and reworking this HTML later on when we start to build Marionette views. It is a good idea, however, to keep these demo HTML pages within your project, so that you can test their look and feel on different browsers and devices, all the while tweaking your HTML layout and CSS styles.

Data structure

In a real-world application, data for websites would be stored and retrieved from a database of some sort. To use the data within a JavaScript web page, these data structures would be serialized to JSON format. Marionette uses standard Backbone models and collections for loading and serializing data structures. For the purpose of this sample application, our data structure will look like this:



Class diagram of ManufacturerCollection and related Backbone models

The source of our data is the `ManufacturerCollection`, which will have a `url` property to load data from our site. This `ManufacturerCollection` holds a collection of `ManufacturerModels`, that are available via the `models` property. The `ManufacturerCollection` also implements two interfaces: `IManufacturerCollection` and `IFilterProvider`. We will discuss these two interfaces later on.

The properties of the `ManufacturerModel` will be used to render a single manufacturer's name and logo to the DOM. Each `ManufacturerModel` also has an array named `boards`, which holds an array of `BoardModels`.

Each `BoardModel` has properties that are necessary for rendering, as well as an array named `board_types`, which holds an array of `BoardType` classes. A `BoardType` is a simple string, and will hold a value of either "Wave", "Freestyle", or "Slalom".

Each `BoardModel` will also have an array of `sizes`, holding a `BoardSize` class, containing detailed information on the available sizes.

As an example, the JSON data structure that is used to serialize the preceding object structure, would be as follows:

```
{
  "manufacturer": "JP Australia",
  "manufacturer_logo": "jp_australia_logo.png",
  "logo_class": "",
  "boards": [
    {
      "name": "Radical Quad",
      "board_types": [ { "board_type": "Wave" } ],

      "description": "Radical Wave Board",
      "image": "jp_windsurf_radicalquad_ov.png",
      "long_description": "long desc goes here",
      "sizes": [
        { "volume": 68, "length": 227,
          "width": 53, "sail_min": "< 5.0", "sail_max": "< 5.2" }
      ]
    }
  ]
}
```

In our sample application, a full JSON dataset can be found at `/tscode/tests/boards.json`.

Data interfaces

In order to use this JSON data structure within TypeScript, we will need to define a set of interfaces to describe the above data structure, as follows:

```
export interface IBoardType {
  board_type: string;
}
export interface IBoardSize {
  volume: number;
  length: number;
  width: number;
  sail_min: string;
  sail_max: string;
}
```

```
export interface IBoardModel {
  name: string;
  board_types: IBoardType[];
  description: string;
  image: string;
  long_description: string;
  sizes: IBoardSize[];
}
export interface IManufacturerModel {
  manufacturer: string;
  manufacturer_logo: string;
  logo_class: string;
  boards: IBoardModel[];
}
```

These interfaces simply match the model properties in the previous diagram, and we can then build the corresponding `Backbone.Model` classes that implement these interfaces. Note that for brevity, we have not listed each individual property of each model here, so be sure to refer to the accompanying source code for a full listing. Our Backbone models are as follows:

```
export class BoardType extends Backbone.Model
  implements IBoardType {
  get board_type() { return this.get('board_type'); }
  set board_type(val: string) { this.set('board_type', val); }
}
export class BoardSize extends Backbone.Model
  implements IBoardSize {
  get volume() { return this.get('volume'); }
  set volume(val: number) { this.set('volume', val); }
  // more properties
}
export class BoardModel extends Backbone.Model implements IBoardModel
{
  get name() { return this.get('name'); }
  set name(val: string) { this.set('name', val); }
  // more properties
  get sizes() { return this.get('sizes'); }
  set sizes(val: IBoardSize[]) { this.set('sizes', val); }
}
export class ManufacturerModel extends Backbone.Model implements
  IManufacturerModel {
  get manufacturer() { return this.get('manufacturer'); }
}
```

```
    set manufacturer(val: string) { this.set('manufacturer', val); }
    // more properties
    get boards() { return this.get('boards'); }
    set boards(val: IBoardModel[]) { this.set('boards', val); }
  }
```

Each class extends `Backbone.Model`, and implements one of the interfaces that we have defined earlier. There is not much to these classes, except for defining a `get` and `set` method for each property, and using the correct property type.

At this stage, our models are in place, and we can write a few unit tests, just to make sure that we can create our models correctly:

```
it("should build a BoardType", () => {
  var boardType = new bm.BoardType(
    { board_type: "testBoardType" });
  expect(boardType.board_type).toBe("testBoardType");
});
```

We start with a simple test that creates a `BoardType` model, and then test that the `board_type` property has been set correctly. Similarly, we can create a test for the `BoardSize` model:

```
describe("BoardSize tests", () => {
  var boardSize: bm.IBoardSize;
  beforeEach(() => {
    boardSize = new bm.BoardSize(
      { "volume": 74, "length": 227,
        "width": 55, "sail_min": "4.0", "sail_max": "5.2" });
  });
  it("should build a board size object", () => {
    expect(boardSize.volume).toBe(74);
  });
});
```

This test is also just creating an instance of the `BoardSize` model, but it is using the `beforeEach` Jasmine method. For brevity, we are only showing one test, which checks the `volume` property, but in a real-world application we would test each of the `BoardSize` properties. Finally, we can write a test of the `BoardModel` as follows:

```
describe("BoardModel tests", () => {
  var board: bm.IBoardModel;
  beforeEach(() => {
    board = new bm.BoardModel({
```

```

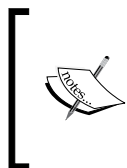
        "name": "Thruster Quad",
        "board_types": [{ "board_type": "Wave" }],
        "description": "Allround Wave Board",
        "image": "windsurf_thrusterquad_ov.png",
        "long_description":
            "Shaper Werner Gnigler and pro riders Robby Swift",
        "sizes": [
            { "volume": 73, "length": 228, "width": 55.5,
              "sail_min": "4.0", "sail_max": "5.2" }
        ]
    });
});

it("should find name property", () => {
    expect(board.name).toBe("Thruster Quad");
});
it("should find sizes[0].volume property", () => {
    expect(board.sizes[0].volume).toBe(73);
});
it("should find sizes[0].sail_max property", () => {
    expect(board.sizes[0].sail_max).toBe("5.2");
});
it("should find board_types[0].sail_max property", () => {
    expect(board.board_types[0].board_type).toBe("Wave");
});
});

```

Again, we are creating a `BoardModel` instance in our `beforeAll` function, and then testing that the properties are set correctly. Note the tests near the bottom of this code snippet: we are checking whether the `sizes` property and `board_types` properties have been built correctly, and that they are in fact arrays that can be referenced with `[]` array notation.

In the accompanying source code, you will find further tests for these models, as well as tests for the `ManufacturerModel`.



Note how each model is constructed with a simple cut-and-paste of sections of the original JSON sample. When Backbone models are hydrated through RESTful services, these services are simply returning JSON—and our tests are, therefore, matching what Backbone itself would be doing.

Integration tests

At this stage, you may wonder why we are writing these sort of tests, as they might seem trivial, and are just checking whether certain properties have been constructed correctly. In real-world applications, models change quite frequently, especially in the beginning stages of a project. It is quite common to have one developer, or a portion of the team, who are responsible for the backend databases and server-side code that deliver JSON to the frontend. Another another team may be responsible for working on the frontend JavaScript code. By writing tests like these, you are clearly defining what your data structures should look like, and what properties you are expecting in your models. If a change is made server side that modifies a data structure, your team will be able to quickly identify where the cause of the problem lies.

Another reason to write property-based tests is that Backbone, Marionette, and just about any other JavaScript library will use these property names to render HTML to the frontend. If you have a template that is expecting a property called `manufacturer_logo`, and you change this property name to `logo_image`, then your rendering code will break. These errors are quite often difficult to track down at runtime. Following the Test Driven Development mantra of "fail early, and fail loudly", our model property tests will quickly highlight these potential errors, should they occur.

Once a series of property-based tests are in place, we can now focus on an integration test that will actually call the server-side code. This will ensure that our RESTful services are working correctly, and that the JSON data structure that our site is generating matches the JSON data structure that our Backbone models expect. Again, if two separate teams are responsible for client-side and server-side code, this sort of integration test will ensure that the data exchange is consistent.

We will be loading our data for this application through a `Backbone.Collection` class, and this collection will need to load multiple manufacturers. To this end, we can now build a `ManufacturerCollection` class as follows:

```
export class ManufacturerCollection
  extends Backbone.Collection<ManufacturerModel>
{
  model = ManufacturerModel;
  url = "/tscode/boards.json";
}
```

This is a very simple `Backbone.Collection` class, which just sets the `model` property to our `ManufacturerModel`, and the `url` property to `/tscode/boards.json`. As our sample application does not have a backend database or REST services, so we will just load our JSON from disk at this stage. Note that even though we are using a static JSON file in this test, Backbone will still issue an HTTP request back to our server in order to load this file, meaning that any test of this `ManufacturerCollection` is, in fact, an integration test. We can now write some integration tests to ensure that this model can be loaded correctly from the `url` property, as follows:

```
describe("ManufacturerCollection tests", () => {
  var manufacturers: bm.ManufacturerCollection;

  beforeEach(() => {
    manufacturers = new bm.ManufacturerCollection();
    manufacturers.fetch({ async: false });
  });

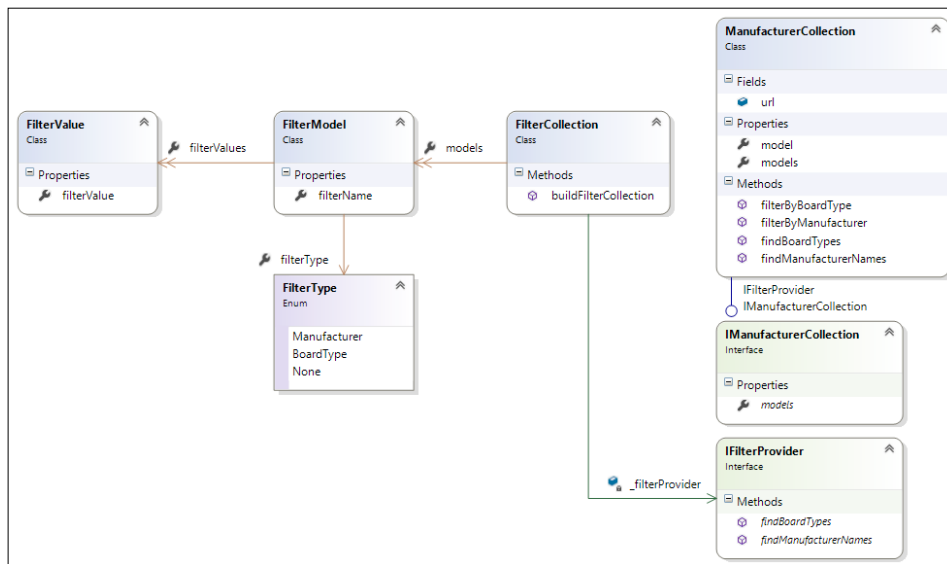
  it("should load 3 manufacturers", () => {
    expect(manufacturers.length).toBe(3);
  });

  it("should find manufacturers.at(2)", () => {
    expect(manufacturers.at(2).manufacturer)
      .toBe("Starboard");
  });
})
```

We are again using the Jasmine `beforeAll` syntax to set up our `ManufacturerCollection` instance, and then calling `fetch({ async: false })` to wait for the collection to be loaded. We then have two tests, one to check that we are loading three manufacturers into our collection, and another to check the `Manufacturer` model at index 2.

Traversing a collection

Now that we have a full `ManufacturerCollection` loaded, we can turn our attention to processing the data that it contains. We will need to search this collection to find two things: a list of manufacturers, and a list of board types. These two lists will be used by our filtering panel on the left-hand side panel. In a real-world application, these two lists may be provided by server-side code, returning simple JSON data structures to represent these two lists. In our sample application, however, we will show how to traverse the main manufacturer Backbone collection that we have already loaded. The filtering data structure is as follows:



FilterCollection class diagram with related Backbone models

Rather than listing the full implementation of Backbone models shown in the preceding diagram, we will take a look at the TypeScript interfaces instead. Our interfaces for these filtering models are as follows:

```
export enum FilterType {  
    Manufacturer,  
    BoardType,  
    None  
}
```

```
}  
export interface IFilterValue {  
    filterValue: string;  
}  
export interface IFilterModel {  
    filterType: FilterType;  
    filterName: string;  
    filterValues: IFilterValue[];  
}
```

We start with a `FilterType` enum, which we will use to define each of the types of filters we have available. We can filter our board list by either manufacturer name, board type, or clear all filters by using the `None` filter type.

The `IFilterValue` interface simply holds a string value that will be used for filtering. When we are filtering by board type, this string value would be one of "Wave", "Freestyle", or "Slalom", and when we are filtering by manufacturer, this string value will be the name of the manufacturer.

The `IFilterModel` interface will hold the `FilterType`, a name for the filter, and array of `filterValues`.

We will create a Backbone model for each of these interfaces, meaning that we will end up with two Backbone models, named `FilterValue` (which implements the `IFilterValue` interface), and `FilterModel` (which implements the `IFilterModel` interface). To house a collection of `FilterModel` instances, we will also create a Backbone collection named `FilterCollection`. This collection has a single method named `buildFilterCollection`, which will use an `IFilterProvider` interface to build its internal array of `FilterModels`. This `IFilterProvider` interface is as follows:

```
export interface IFilterProvider {  
    findManufacturerNames(): bm.IManufacturerName[];  
    findBoardTypes(): string[]  
}
```

Our `IFilterProvider` interface has two functions. The `findManufacturerNames` function will return a list of manufacturer names (and their associated logos), and the `findBoardTypes` function will return a list of strings of all board types. This information is all that is needed to build up our `FilterCollection` internal data structures.

All of the values needed to populate this `FilterCollection` will come from data that is already contained within our `ManufacturerCollection`. The `ManufacturerCollection` will, therefore, need to implement this `IFilterProvider` interface.

Finding manufacturer names

Let's continue working within our test suite to flesh out the functionality of the `findManufacturerNames` function that the `ManufacturerCollection` will need to implement, as part of the `IFilterProvider` interface. This function returns an array of type `IManufacturerName`, which is defined as follows:

```
export interface IManufacturerName {
  manufacturer: string;
  manufacturer_logo: string;
}
```

We can now build a test using this interface:

```
it("should return manufacturer names ", () => {
  var results: bm.IManufacturerName[] =
    manufacturers.findManufacturerNames();
  expect(results.length).toBe(3);
  expect(results[0].manufacturer).toBe("JP Australia");
});
```

This test is reusing the `manufacturers` variable that we set up in our previous test suite. It then calls the `findManufacturerNames` function, and expects the results to be an array of three manufacturer names, i.e. "JP Australia", "RRD", and "Starboard".

Now, we can update the actual `ManufacturerCollection` class, in order to provide an implementation of the `findManufacturerNames` function:

```
public findManufacturerNames(): IManufacturerName[] {
  var items = _(this.models).map((iterator) => {
    return {
      'manufacturer': iterator.manufacturer,
      'manufacturer_logo': iterator.manufacturer_logo
    };
  });
  return items;
}
```

In this function, we are using the Underscore utility function named `map` to loop through our collection. Each Backbone collection class has an internal array named `models`. The `map` function will loop through this `models` property, and call the anonymous function for each item in the collection, passing the current model into our anonymous function via the `iterator` argument. Our code then builds a JSON object with the required properties of the `IManufacturer` interface.



The TypeScript compiler will generate errors if the returned object does not conform to the `IManufacturer` name interface.

Finding board types

We can now focus on the second function of the `IFilterProvider` interface, named `findBoardTypes` that the `ManufacturerCollection` will need to implement. Here is the unit test:


```
it("should find board types ", () => {
  var results: string[] = manufacturers.findBoardTypes();
  expect(results.length).toBe(3);
  expect(results).toContain("Wave");
  expect(results).toContain("Freestyle");
  expect(results).toContain("Slalom");
});
```

This test calls the `findBoardTypes` function, which will return an array of strings. We are expecting the returned array to contain three strings: "Wave", "Freestyle", and "Slalom".

The corresponding function in our `ManufacturerCollection` class is then implemented as follows:

```
public findBoardTypes(): string[] {
  var boardTypes = new Array<string>();
  _(this.models).each((manufacturer) => {
    _(manufacturer.boards).each((board) => {
      _(board.board_types).each((boardType) => {
        if (!_.contains(
          boardTypes, boardType.board_type)) {
          boardTypes.push(boardType.board_type);
        }
      });
    });
  });
  return boardTypes;
}
```

The implementation of the `findBoardTypes` function starts by creating a new string array named `boardTypes`, which will hold our results. We then use the Underscore `each` function to loop through each manufacturer. The Underscore `each` function is similar to the `map` function, and will iterate through each item in our collection. We then loop through each board in the manufacturer's arsenal, and through each board type listed per board. Finally, we are testing to see whether the board type collection contains an item already, using the underscore `_.contains` function. If it does not already have the board type in the array, we push the `board_type` string into our `boardTypes` array.

 The Underscore library has numerous utility functions available for searching, manipulating, and modifying arrays and collections – so be sure to consult the documentation to find suitable functions for use in your code. These functions are not limited to Backbone collections only, and can be used on any type of array.

This completes our work on the `IFilterProvider` interface, and its implementation within the `ManufacturerCollection` class.

Filtering a Collection

When a user clicks on a filter option on the left-hand side panel, we will need to apply the selected filter to the data contained within our manufacturer collection. In order to do this, we will need to implement two functions, named `filterByManufacturer`, and `filterByBoardType` within the `ManufacturerCollection` class. Let's start with a test to filter our collection by manufacturer name:

```
it("should filter by manufacturer name ", () => {
  var results = manufacturers.filterByManufacturer("RRD");
  expect(results.length).toBe(1);
});
```

This test calls the `filterByManufacturer` function, expecting only a single manufacturer to be returned. With this test in place, we can create the real `filterByManufacturer` function on the `ManufacturerCollection` as follows:

```
public filterByManufacturer(manufacturer_name: string) {
  return _(this.models).filter((item) => {
    return item.manufacturer === manufacturer_name;
  });
}
```

Here, we are using the Underscore function named `filter` to apply a filter to our collection.

The second filtering function is by board type, and is a little more complicated. We will need to loop through each manufacturer in our collection, then through each board, and then through each board type. If we find a match for the board type, we will flag this board to be included in the result set. Before we tackle the `filterByBoardType` function, let's write a test:

```
it("should only return Slalom boards ", () => {
  var results = manufacturers.filterByBoardType("Slalom");
  expect(results.length).toBe(2);
  _(results).each((manufacturer) => {
    _(manufacturer.boards).each((board) => {
      expect(_(board.board_types).some((boardType) => {
        return boardType.board_type == 'Slalom';
      })).toBeTruthy();
    });
  });
});
```

Our test calls the `filterByBoardType` function, using the string "Slalom" as a filter. Remember that this function will return a collection of `ManufacturerModel` objects at the top level, with the `boards` array within each of these objects filtered by board type. Our test then loops through each manufacturer, and each board in the result set, and then uses the Underscore function called `some` to test whether the `board_types` array has the correct board type.

Our code to implement this function on the `ManufacturerCollection` is also a little tricky, as follows:

```
public filterByBoardType(board_type: string) {
  var manufWithBoard = new Array();
  _(this.models).each((manuf) => {
    var hasBoardtype = false;
    var boardMatches = new Array();
    _(manuf.boards).each((board) => {
      var match = _(board.board_types).some((item) => {
        return item.board_type == board_type;
      });
      if (match) {
        boardMatches.push(new BoardModel(board));
        hasBoardtype = true;
      }
    });
  });

  if (hasBoardtype) {
```



```
        var manufFiltered = new ManufacturerModel(manuf);
        manufFiltered.set('boards', boardMatches);
        manufWithBoard.push(manufFiltered);
    }
    });
    return manufWithBoard;
}
```

Our `ManufacturerCollection` class instance holds the entire collection that was loaded via the JSON file from the site. In order to keep this data for repeated filters, we will need to construct a new `ManufacturerModel` array to return from this function – so that we do not to modify the underlying "global" data. Once we have constructed this new array, we can then loop through each manufacturer. If we find a board matching the required filter, we will set a flag named `hasBoardType` to true, to indicate that this manufacturer must be added to our filtered array.

Each manufacturer in this filtered array will also need to list only the board types that match our filter criteria, so we will need another array – called `boardMatches` – to hold these matching boards. Our code will then loop through each board, and check whether it has the required `board_type`. If so, we will add it to the `boardMatches` array and set the `hasBoardType` flag to true.

Once we have looped through each board for a manufacturer, we can check the `hasBoardType` flag. If our manufacturer has this board type, we will construct a new `ManufacturerModel`, and then set the `boards` property on this model to our in-memory array of the matching boards.

Our work with the underlying Backbone collections and models is now complete. We have also written a set of unit and integration tests to ensure that we can load our collection from the site, build our filtering lists from this collection, and then apply a particular filter to this data.

Marionette application, regions and layouts

We can now focus our attention on building the application itself. In Marionette, this is achieved by creating a class that derives from `Marionette.Application`, as follows:

```
export class BoardSalesApp extends Marionette.Application {
  viewLayout: pvl.PageViewLayout;
  constructor(options?: any) {
    if (!options)
```

```
        options = {};  
        super();  
        this.viewLayout = new pvl.PageViewLayout();  
    }  
    onStart() {  
        this.viewLayout.render();  
    }  
}
```

Here, we have defined a class named `BoardSalesApp` that derives from the `Marionette.Application` class, and will serve as the starting point for our application. Our constructor function is fairly simple, and creates a new instance of the `PageViewLayout` class, which we will discuss shortly. The only other function in our application is the `onStart` function, which renders our `PageViewLayout` to the screen. This `onStart` function will be triggered by Marionette when the application starts.

Our `PageLayoutView` class is as follows:

```
export class PageViewLayout extends  
    Marionette.LayoutView<Backbone.Model> {  
    constructor(options?: any) {  
        if (!options)  
            options = {};  
        options.el = '#page_wrapper';  
        var snippetService: ISnippetService =  
            TypeScriptTinyIoC.resolve(IISnippetService);  
        options.template = snippetService.retrieveSnippet(  
            SnippetKey.PAGE_VIEW_LAYOUT_SNIPPET);  
        super(options);  
    }  
}
```

This class extends from `Marionette.LayoutView`, and does two important things. Firstly, it sets a number of properties on the `options` object, and then calls the base class constructor via the `super` function, passing in this `options` object. One of the properties of this `options` object is named `el`, and contains the name of the DOM element that this view will render into. In this code snippet, this `el` property is set to the DOM element `'#page_wrapper'`. Without this `el` property set, we will just get a blank screen when we try to render the view to the DOM.

The second important step in our constructor is to load a snippet from the `SnippetService`. This snippet is then used to set the `template` property on the `options` object. Marionette, similar to Backbone, loads a template, and then combines the underlying model properties with the view template, in order to generate the HTML that will be rendered to the DOM.

At this stage, in order to run our `BoardSalesApp`, and have it render the `PageViewLayout` to the DOM, we will need two things. The first is a DOM element in our `index.html` page with an `id="page_wrapper"`, to match our `options.el` property, and the second is our `PAGE_VIEW_LAYOUT_SNIPPET`.

Our `index.html` page is as follows:

```
<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml" >
<head>
  <title>BoardSales</title>
  <link rel="stylesheet" href="/Content/bootstrap.css" />
  <link rel="stylesheet" type="text/css"
    href="/Content/app.css">
  <script type="text/javascript"
    src="/Scripts/head-1.0.3.js"></script>
  <script data-main="/tscode/app/AppConfig"
    type="text/javascript"
    src="/Scripts/require.js"></script>
</head>
<body>
  <div id="page_wrapper">

  </div>
  <footer class="footer footer_style">
    <div class="container">
      <p class="text-muted"><small>Footer</small></p>
    </div>

  </footer>
</body>
</html>
```

This page includes the `bootstrap.css` and `app.css` style sheets, as well as a call to `Require`, with the `data-main` attribute set to a `Require` config file named `/tscode/app/AppConfig`. The body of the `index.html` page just includes the DOM element with `id="page_wrapper"`, and a footer. This is a very stripped-down version of the demo HTML page that we built earlier.



We have also included a script named `head-1.0.3.js`, which can be installed via the NuGet package `HeadJS`. This script interrogates our browser to find out whether it is running on a mobile or desktop device, what browser we are using, and even what the current screen size is. We will use the output of `head.js` later in our application.

We will now need to create an HTML snippet for the `PageViewLayout`. This file is called `PageViewLayout.html`, and resides in the `/tscode/app/views` directory, so it can be easily found when we are working with the `PageViewLayout.ts` file. Take a look at the sample code for a full listing of this HTML file, which includes the following relevant sections:

```
<div id="page_wrapper">
  <div id="main_panel_div">
    <div class="carousel-inner" >
      <div id="carousel_panel_1" >
        <div id="content_panel_left" >
          <!--filter panel goes here-->
        </div>
        <div id="content_panel_main">
          <div id="manufacturer_collection">
            <!--board list goes here-->
          </div>
        </div>
      </div>
      <div id="carousel_panel_2">
        <!--board detail panel goes here-->
      </div>
    </div>
  </div>
</div>
```

Our `PageViewSnippet.html` file contains the major elements of our page. We have the `main_panel_div` that serves as the middle panel of our application, with a `carousel-inner` div that contains our two carousel panel divs, named `carousel_panel_1` and `carousel_panel_2`. Within these carousel panels, we will be rendering the filter panel, board list panel and board detail panel.

We now need to put together our `AppConfig.ts` file that `Require` will load, and set up the `SnippetService` to load the `PageViewLayout.html` snippet. In the interests of brevity, we have not listed the full `require.config` here, and have excluded the paths and shims section. We will just focus on the call to `Require` as follows:

```
require([
  'BoardSalesApp', 'tinyioc', 'snippetService'
  , 'text!/tscode/app/views/PageViewLayout.html' ],
  (app, tinyioc, snippetService, pageViewLayoutSnippet) => {

  var snippetService = new SnippetService();
  snippetService.storeSnippet(
    SnippetKey.PAGE_VIEW_LAYOUT_SNIPPET,
```

```
        pageViewLayoutSnippet);
    TypeScriptTinyIoC.register(snippetService, IISnippetService);

    var boardSalesApp = new app.BoardSalesApp();
    boardSalesApp.start();

});
```

Here, we included the `BoardSalesApp`, `tinyioc`, and `snippetService`, as well as our `PageViewLayout.html` snippet in the call to `require`. We then set up the `SnippetService`, store the `pageViewLayoutSnippet` against the correct key, and register the `SnippetService` with our service locator. To start our Marionette application, we create a new instance of the `BoardSalesApp`, and call `start`. Once the `start` method is called, our `BoardSalesApp.onStart` method will be fired by Marionette, which will then render the `PageViewLayout` class.

Loading the main collection

In this application, we will be loading our `ManufacturerCollection` only once, and then reusing this "global" collection for filtering purposes. Let's now update our `BoardSalesApp` to include this "global" collection, and load it on application startup. Again, refer to the sample code for a full listing:

```
export class BoardSalesApp extends Marionette.Application {
  viewLayout: pvl.PageViewLayout;
  _manufCollection: bm.ManufacturerCollection;

  constructor(options?: any) {
    if (!options)
      options = {};
    super();
    _.bindAll(this, 'CollectionLoaded');
    _.bindAll(this, 'CollectionLoadError');
    this.viewLayout = new pvl.PageViewLayout();
  }

  onStart() {
    this.viewLayout.render();
    this._manufCollection = new bm.ManufacturerCollection();
    TypeScriptTinyIoC.register(this._manufCollection,
      bm.IManufacturerCollection);
    this._manufCollection.fetch({
      success: this.CollectionLoaded,
      error: this.CollectionLoadError });
  }
}
```

```

    }

    CollectionLoaded() {
        TypeScriptTinyIoC.raiseEvent (
            new ev.NotifyEvent (
                ev.EventType.ManufacturerDataLoaded),
            ev.IINotifyEvent);
    }

    CollectionLoadError(err) {
        TypeScriptTinyIoC.raiseEvent (
            new ev.ErrorEvent (err), ev.IIErrorEvent);
    }
}

```

We have updated our `BoardSalesApp` to store an instance of the `ManufacturerCollection` class in the private variable named `_manufCollection`. Our `onStart` function has been updated to instantiate this collection, after the call to `viewLayout.render`. Note the next call to `TypeScriptTinyIoC`. We are registering this `_manufCollection` as a service that will implement the `IManufacturerCollection` named interface. We then call the Backbone `fetch` function on the collection, with a success and error callback. Both the success callback and error callback simply raise an event.

By registering our instance of the `ManufacturerCollection` class against the named interface `IManufacturerCollection`, any of our classes that need access to the main collection can simply request the instance of this class from our service locator. These named interfaces are as follows:

```

export interface IManufacturerCollection {
    models: ManufacturerModel[];
}
export class IManufacturerCollection implements IInterfaceChecker {
    propertyNames = ['models'];
    className = 'IManufacturerCollection';
}

```

We will also need to modify our `ManufacturerCollection` class to implement the `IManufacturerCollection` interface as follows:

```

export class ManufacturerCollection extends
    Backbone.Collection<ManufacturerModel>
    implements IManufacturerCollection
{
    // existing code
}

```

Let's now have a look at the events that will be fired from our success and error callbacks. In the success function callback, we are raising an event of type `INotifyEvent`. Note that we are just listing the interface definitions here—for the corresponding `IInterfaceChecker` classes and event classes, please refer to the accompanying source code:

```
export enum EventType {
  ManufacturerDataLoaded,
  ErrorEvent
}
export interface INotifyEvent {
  eventType: EventType;
}
export interface INotifyEvent_Handler {
  handle_NotifyEvent(event: INotifyEvent): void;
}
```

Here, we have defined an `EventType` enum to hold an event type, and then defined an `INotifyEvent` interface that just holds a property named `eventType`. We have also defined the corresponding `INotifyEvent_Handler` interface that any handler will need to implement.

Our error event will use inheritance to derive from these interfaces as follows:

```
export interface IErrorEvent extends INotifyEvent {
  errorMessage: string;
}
export interface IErrorEvent_Handler {
  handle_ErrorEvent(event: IErrorEvent);
}
```

Here, we are deriving the `IErrorEvent` interface from `INotifyEvent`, thereby reusing the `EventType` enum and properties from the base interface.

We can now respond to these events in our `PageViewLayout` class:

```
export class PageViewLayout extends
  Marionette.LayoutView<Backbone.Model>
  implements ev.INotifyEvent_Handler
{
  private _manufacturerView: mv.ManufacturerCollectionView;
  constructor(options?: any) {
    // existing code
    _.bindAll(this, 'handle_NotifyEvent');
    TypeScriptTinyIoC.registerHandler(
      this, ev.IINotifyEvent_Handler, ev.IINotifyEvent);
  }
  handle_NotifyEvent(event: ev.INotifyEvent) {
```

```

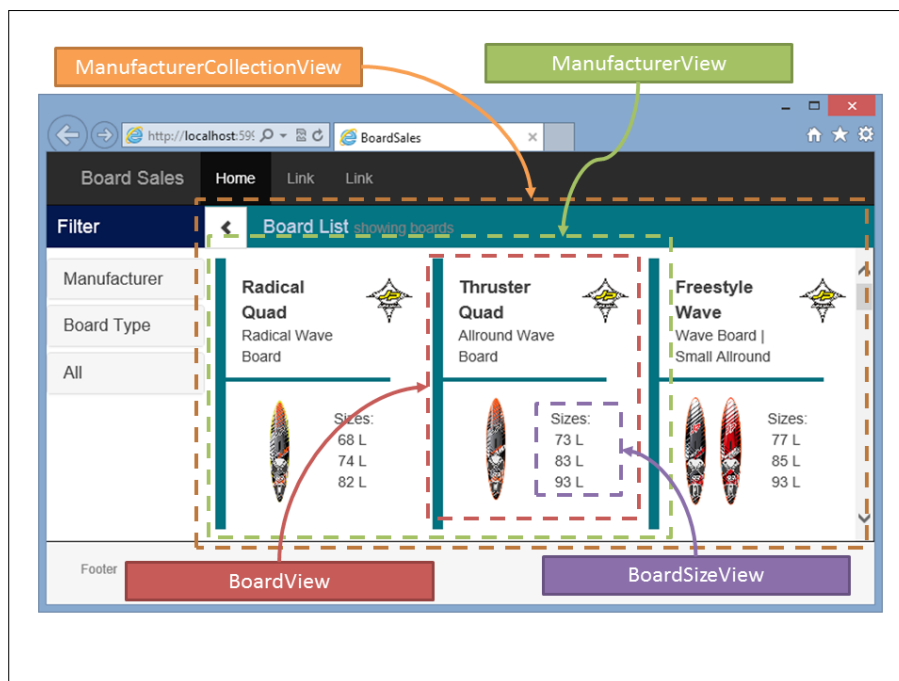
    if (event.eventType == ev.EventType.ManufacturerDataLoaded)
    {
        this._manufacturerView =
            new mv.ManufacturerCollectionView();
        this._manufacturerView.render();
    }
}
}

```

We have implemented the `INotifyEvent_Handler` interface, and registered with `TypeScriptTinyIoC` for the `INotifyEvent`. Our `handle_NotifyEvent` class will check that the event type is a `ManufacturerDataLoaded` event, and then create an instance of the `ManufacturerCollectionView` class and render it to the DOM.

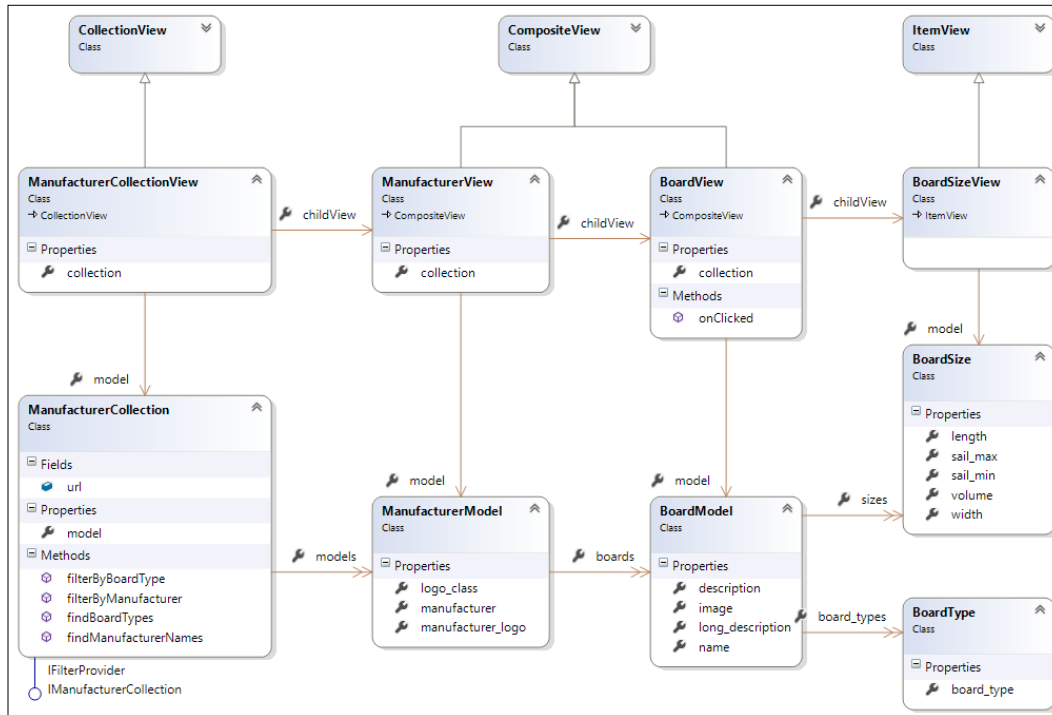
Marionette views

Marionette provides a number of different view classes for us to use, based on what type of object we need to render to the DOM. Any class that needs to render a `Backbone.Collection` can use a `CollectionView`, and any class that needs to render a single item in this collection can use an `ItemView`. Marionette also provides a hybrid of these two views called a `CompositeView`. If we take a look at our demo application, we will be able to break up our screen into a number of logical views, as follows:



Board list view with Marionette view overlay

The identification of what views we need to build are pretty closely related to the data structure that we have in place for our Backbone collections and models. This relationship is clearly seen when we superimpose the preceding views on top of our class diagram for our `ManufacturerCollection`:



Model class diagram with corresponding Marionette Views

The ManufacturerCollectionView class

We start with a `ManufacturerCollectionView`, which is a view that renders the whole `ManufacturerCollection`. We will also need a `ManufacturerView` to render a particular `ManufacturerModel`, and then a `BoardView` to render each board in a manufacturer's arsenal. Each board has an internal array of `BoardSize` objects, so we will create a `BoardSizeView` to render these items.

Lets start building these views, starting with the `ManufacturerCollectionView`:

```
export class ManufacturerCollectionView
  extends Marionette.CollectionView<bm.ManufacturerModel> {
  constructor(options?: any) {
    if (!options)
      options = {};
```

```
options.el = '#manufacturer_collection';
options.className = "row board_row";

super(options);
this.childView = ManufacturerView;

var manufColl: bm.IManufacturerCollection =
  TypeScriptTinyIoC.resolve(bm.IIManufacturerCollection);
if (!options.collection) {
  this.collection =
    <Backbone.Collection<bm.ManufacturerModel>> manufColl;
} else {
  this.collection = options.collection;
}
}
```

This class extends from `Marionette.CollectionView`, and specifies our `ManufacturerModel` as the generic type for the class. Our constructor sets the `el` property of the `options` object to be `"#manufacturer_collection"`. As we saw with our `PageLayoutView`, Marionette will use this property to render the entire collection into the DOM. We have also set a `className` property in our `options`. Marionette will use the `className` property to append a `class="..."` attribute to the outer DOM element. This will apply the CSS styles of `row` and `board_row` to the `manufacturer_collection` element in the rendered HTML. Once we have constructed our `options` correctly, we call `super(options)` to pass these options to the base class constructor.

The `childView` property of a `CollectionView` instructs Marionette to create an instance of the class we specify for each element that it finds in the collection. We have set this `childView` property to be `ManufacturerView`, so Marionette will construct a new `ManufacturerView` for each element in the collection.

Finally, in our constructor, we are using our service locator pattern to look up an instance of our `ManufacturerCollection` service, and then we set the internal `this.collection` property to the returned object. Once we have defined a `childView` class name, and set the `this.collection` property, Marionette will automatically create instances of our child views, and render them to the DOM.

Note that we don't need an HTML template or snippet for a `CollectionView`. This is because we are deferring the rendering of an individual item to the `childView` classes.

The ManufacturerView class

Our `childView` class, `ManufacturerView`, is as follows:

```
export class ManufacturerView
  extends Marionette.CompositeView<Backbone.Model> {
  constructor(options?: any) {
    if (!options)
      options = {};
    options.template = _.template('<div></div>');
    super(options);
    this.collection = new Backbone.Collection(
      this.model.get('boards')
    );
    this.childView = BoardView;
    this.childViewOptions = {
      parentIcon: this.model.get('manufacturer_logo')
    };
  }
}
```

In this instance, we are deriving our view from `Marionette.CompositeView`, and using a standard `Backbone.Model` for the generic type. Because we have multiple manufacturers in our board list view, we don't really need to render anything specific for each manufacturer. Therefore, our template is a simple `<div></div>`.

The important part of this view is to set up a new `Backbone.Collection` for our boards array, and then set a `childView` class to render each board in the collection. Our `childView` property is set to `BoardView`, and we are also setting a `childViewOptions` property that will be sent through to each `BoardView` instance. Remember that each `BoardView` shows the manufacturer logo, but this logo image is held at the manufacturer level. Therefore, we need to pass this information down to each `BoardView` that is created. `Marionette` allows us to use the `childViewOptions` property to pass any extra properties down to the child view. Here, we have defined a `parentIcon` property as part of this `childViewOptions` object, in order to pass down the manufacturer logo to each instance of a child `BoardView` class. This `parentIcon` property will then be available to the child view via the `options` parameter.

The BoardView class

Our `BoardView` class is also a `CompositeView` as follows:

```
export class BoardView
  extends Marionette.CompositeView<bm.BoardModel> {
```

```
constructor(options?: any) {
  if (!options)
    options = {};
  var snippetService: ISnippetService =
    TypeScriptTinyIoC.resolve(IISnippetService);
  options.template = _.template(
    snippetService.retrieveSnippet(
      SnippetKey.BOARD_VIEW_SNIPPET)
  );
  super(options);

  this.model.set('parentIcon', options.parentIcon);

  this.collection =
    <any>(new Backbone.Collection(
      this.model.get('sizes')));
  this.childView = BoardSizeView;
  this.childViewContainer = 'tbody';

  var snippetService: ISnippetService =
    TypeScriptTinyIoC.resolve(IISnippetService);
  this.childViewOptions = {
    template: _.template(
      snippetService.retrieveSnippet(
        SnippetKey.BOARD_SIZE_MINI_VIEW_SNIPPET)
    )
  };
}
}
```

This BoardView constructor does a couple of things. Firstly, it retrieves the snippet named `BOARD_VIEW_SNIPPET` to use as its own `template`. It then sets an internal model property named `parentIcon` to store the `parentIcon` property that was passed in via the `options` parameter from the parent view. We then create a new `Backbone.Collection` for the `sizes` array, and set the `childView` property to `BoardSizeView`. The `childViewContainer` property tells Marionette that there is a `<tbody></tbody>` HTML div within our snippet that it should use to render any `childView` into. Finally, we retrieve another snippet named `BOARD_SIZE_MINI_VIEW_SNIPPET`, and pass this snippet through to the `childView` as a `template` property.

Instead of each `BoardSizeView` class resolving its own HTML snippet, we have moved control of which snippet to use up in the class hierarchy, to the parent of the `BoardSizeView`. This allows us to reuse the `BoardSizeView` class within this summary view, as well as in the `BoardDetailView`, which we will discuss later. As the internal data models are identical for the summary size view and the detail size view, all that will need to change is our HTML template. We therefore pass this template down into the `BoardSizeView` using the `childViewOption` properties, as we have seen previously.

The BoardSizeView class

Our `BoardSizeView` class could not be simpler, and is as follows:

```
export class BoardSizeView
  extends Marionette.ItemView<bm.BoardSize> {
  constructor(options?: any) {
    if (!options)
      options = {};
    super(options);
  }
}
```

This class is simply an `ItemView`, which is using the `BoardSize` model as the generic type. We don't have any custom code within this class, but we are simply using it as a named `childView` in our preceding `BoardView` class.

Let's take a look now at the HTML snippets that we will need for each of these views. First up is our `BoardViewSnippet.html`. Again, you can find the full snippet in the accompanying source code. The general structure of the `BoardViewSnippet.html` is as follows:

```
<div class="col-sm-4 board_panel">
  <div class="board_inner_panel">
    <div class="row board_title_row">
      <!-- -some divs just for styling here -->
      <%= name %>
      <!-- -some divs just for styling here -->
      <%= description %>
      
    </div>
    <div class="row board_details_row">
      <a >
        
      </a>
```

```

        <!-- some divs just for styling here -->
        Sizes:
        <table>
            <tbody></tbody>
        </table>
    </div>
</div>
</div>

```

In this snippet, we have included the `<%= name %>`, `<%= description %>`, `<%= parentIcon %>` and `<%= image %>` syntax as placeholders for our model properties. Near the bottom of the snippet, we have created a table with an empty `<tbody></tbody>` tag. This tag corresponds to the `childViewContainer` property that we used in our `BoardView` class, and `Marionette` will render each `BoardSizeView` item into this `<tbody>` tag.

Our `BoardSizeMiniViewSnippet.html` is as follows:

```

<tr>
  <td>&nbsp;</td>
  <td><%= volume %> L</td>
</tr>

```

Here, we are only interested in the `<%= volume %>` property of the `BoardSize` model. With these view classes and two snippets in place, our board list view is complete. All we need to do is to load these snippets up in our `require.config` block, and store the appropriate snippets on our `SnippetService` instance:

```

require([
  'BoardSalesApp', 'tinyioc', 'snippetservice'
  , 'text!./tscode/app/views/PageViewLayout.html'
  , 'text!./tscode/app/views/BoardViewSnippet.html'
  , 'text!./tscode/app/views/BoardSizeMiniViewSnippet.html'
], (app, tinyioc, snippetservice, pageViewLayoutSnippet
  , boardViewSnippet, bsMiniViewSnippet) => {

  var snippetService = new SnippetService();
  snippetService.storeSnippet(
    SnippetKey.PAGE_VIEW_LAYOUT_SNIPPET,
    pageViewLayoutSnippet);
  snippetService.storeSnippet(
    SnippetKey.BOARD_VIEW_SNIPPET, boardViewSnippet);
  snippetService.storeSnippet(
    SnippetKey.BOARD_SIZE_MINI_VIEW_SNIPPET,

```

```
        bsMiniViewSnippet);  
  
        var boardSalesApp = new app.BoardSalesApp();  
        boardSalesApp.start();  
  
    });
```

Filtering using the IFilterProvider interface

When we put together the `ManufacturerCollection` class, we wrote two functions to query the data structure, and return a list of manufacturers and board types. These two functions were called `findManufacturerNames` and `findBoardTypes` respectively. Our new `FilterCollection` class will need to call these methods to retrieve the filter values from our "global" dataset.

We could implement this functionality in two ways. One way would be to get a reference to the global `ManufacturerCollection` instance via the `IManufacturerCollection` named interface. This option, however, would mean that the code for the `FilterCollection` would need to understand the code for the `ManufacturerCollection`. A better way of implementing this functionality would be to get a reference to an `IFilterProvider` interface. This interface would then just expose the two methods that we need to build our list of filters. Let's take this second approach, and define a named interface as follows:

```
export interface IFilterProvider {  
    findManufacturerNames(): bm.IManufacturerName[];  
    findBoardTypes(): string[]  
}  
  
export class IIFilterProvider implements IInterfaceChecker {  
    methodNames = ['findManufacturerNames', 'findBoardTypes'];  
    className = 'IIFilterProvider';  
}
```

We can then simply modify the existing `ManufacturerCollection` to implement this interface (which it already does):

```
export class ManufacturerCollection extends  
    Backbone.Collection<ManufacturerModel>  
    implements IManufacturerCollection, fm.IFilterProvider  
{  
    // existing code  
}
```

We can now register the `ManufacturerCollection` with `TypeScriptTinyIoC` against the `IIFilterProvider` named interface in our `BoardSalesApp.onStart` method, as follows:

```
onStart() {
  this.viewLayout.render();
  this._manufCollection = new bm.ManufacturerCollection();
  TypeScriptTinyIoC.register(this._manufCollection,
    bm.IIManufacturerCollection);
  TypeScriptTinyIoC.register(this._manufCollection,
    fm.IIFilterProvider);
  this._manufCollection.fetch({
    success: this.CollectionLoaded,
    error: this.CollectionLoadError });
}
```

Our `ManufacturerCollection` is now registered to provide both the `IIManufacturerCollection` named interface, as well as the `IIFilterProvider` named interface.

The FilterCollection class

Our `FilterCollection` can then resolve the `IIFilterProvider` interface in its constructor, as follows:

```
export class FilterCollection extends Backbone.Collection<FilterModel>
{
  model = FilterModel;

  private _filterProvider: IIFilterProvider;
  constructor(options?: any) {
    super(options);
    try {
      this._filterProvider =
        TypeScriptTinyIoC.resolve(IIFilterProvider);
    } catch (err) {
      console.log(err);
    }
  }
}
```


Here, we are storing the class that is returned by the call to `TypeScriptTinyIoC` in a private variable named `_filterProvider`. By defining these interfaces for a `FilterProvider`, we can now unit test our `FilterCollection` with a mock `FilterProvider` as follows:

```
class MockFilterProvider implements fm.IFilterProvider {
    findManufacturerNames(): bm.IManufacturerName[] {
        return [
            { manufacturer: 'testManuf1',
              manufacturer_logo: 'testLogo1' },
            { manufacturer: 'testManuf2',
              manufacturer_logo: 'testLogo2' }
        ];
    }
    findBoardTypes(): string[] {
        return ['boardType1', 'boardType2', 'boardType3'];
    }
}
describe('/tscode/tests/models/FilterModelTests', () => {
    beforeAll(() => {
        var mockFilterProvider = new MockFilterProvider();
        TypeScriptTinyIoC.register(
            mockFilterProvider, fm.IIFilterProvider);
    });
});
```

In the setup for our tests, we are creating a `MockFilterProvider` that implements our `IFilterProvider` interface, and we have registered it for the purposes of our tests. By using a mock provider, we also know exactly what data to expect within our tests. Our actual tests will be as follows:

```
describe("FilterCollection tests", () => {
    var filterCollection: fm.FilterCollection;
    beforeAll(() => {
        filterCollection = new fm.FilterCollection();
        filterCollection.buildFilterCollection();
    });

    it("should have two manufacturers", () => {
        var manufFilter = filterCollection.at(0);
```

```

        expect (manufFilter.filterType)
            .toBe (fm.FilterType.Manufacturer);
        expect (manufFilter.filterValues[0].filterValue)
            .toContain ('testManuf1');
    });

    it ("should have two board types", () => {
        var manufFilter = filterCollection.at(1);
        expect (manufFilter.filterType)
            .toBe (fm.FilterType.BoardType);
        expect (manufFilter.filterValues[0].filterValue)
            .toContain ('boardType1');
    });
});

```

These tests start by creating an instance of the `FilterCollectionClass`, and then call the `buildFilterCollection` function. We then test that the collection has a `FilterType.Manufacturer` at index 0, along with expected values. With these failing tests in place, we can flesh out the `buildFilterCollection` function:

```

buildFilterCollection() {
    // build Manufacturer filter.
    var manufFilter = new FilterModel({
        filterType: FilterType.Manufacturer,
        filterName: "Manufacturer"
    });
    var manufArray = new Array<FilterValue>();
    if (this._filterProvider) {
        _(this._filterProvider.findManufacturerNames())
            .each((manuf) => {
                manufArray.push(new FilterValue(
                    { filterValue: manuf.manufacturer }));
            });
        manufFilter.filterValues = manufArray;
    }
    this.push(manufFilter);
    // build Board filter.
    var boardFilter = new FilterModel({
        filterType: FilterType.BoardType,
        filterName: "Board Type"
    });
}

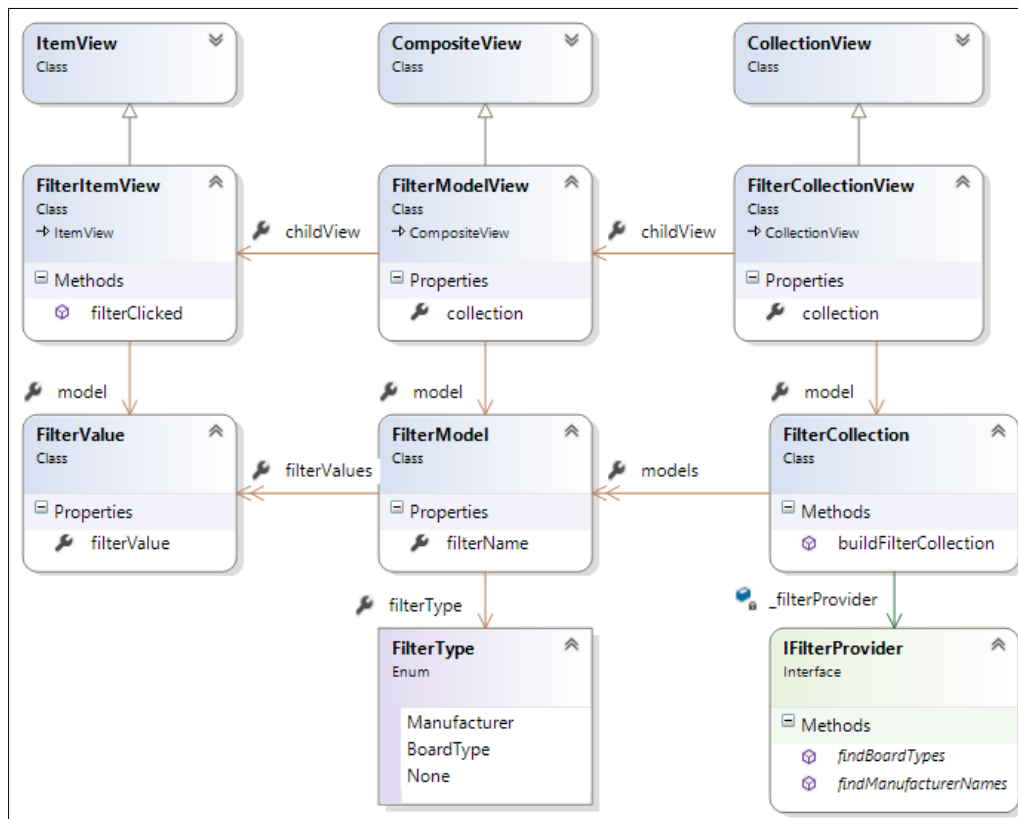
```

```
var boardTypeArray = new Array<FilterValue>();
if (this._filterProvider) {
  _(this._filterProvider.findBoardTypes()).each((boardType) =>
  {
    boardTypeArray.push(new FilterValue(
      { filterValue: boardType }));
  });
  boardFilter.filterValues = boardTypeArray;
}
this.push(boardFilter);
// build All filter to clear filters.
var noFilter = new FilterModel({
  filterType: FilterType.None,
  filterName: "All"
});
var noTypeArray = new Array<FilterValue>();
noTypeArray.push(new FilterValue({ filterValue: "Show All" }));
noFilter.filterValues = noTypeArray;
this.push(noFilter);
}
```

Our `buildFilterCollection` function is creating three instances of a `FilterModel`. The first instance, named `manufFilter` has its `filterType` set to `FilterType.Manufacturer`, and uses the `_filterProvider.findManufacturerNames` function to build up the values for this `FilterModel`. The `manufFilter` instance is then added to the internal collection via the call `this.push(manufFilter)`. The second and third `FilterModel` instances have their `filterType` set to `FilterType.BoardType` and `FilterType.None` respectively.

Filtering views

Again, the relationship between the Marionette views that we need to implement, and the underlying Backbone collections and models that we have, is easy to visualize when we superimpose the views on top of our Backbone models as follows:



Filtering class diagram showing related Marionette views

The first view, named `FilterCollectionView`, will be derived from `CollectionView`, and will be tied to our top-level `FilterCollection`. The second view, named `FilterModelView` will be a `CompositeView`, and will render each `FilterType` to its own accordion header. The third and final view will be an `ItemView` for each of the filter options, and is named `FilterItemView`.

Building these Marionette views is a very similar process to what we have done with the previous manufacturer and board views. For this reason, we will not go into detail here on the implementation of each view. Be sure to refer to the sample code included with this chapter for full listings of these views and their relevant HTML snippets.

Now that we have our filters rendering on the left-hand side panel, we will need to be able to respond to a click event on the `FilterItemView`, and trigger the actual filtering code.

DOM events in Marionette

Marionette provides a simple syntax for trapping DOM events. Any view has an internal property named `events`, which will bind DOM events to our Marionette views. Our `FilterItemView`, then, can be updated to respond to DOM events as follows:

```
export class FilterItemView
  extends Marionette.ItemView<fm.FilterValue> {
  private _filterType: number;
  constructor(options?: any) {
    if (!options)
      options = {};
    options.tagName = "li";
    options.template =
      _.template('<a><%= filterValue %></a>');

    options.events = { click: 'filterClicked' };
    this._filterType = options.filterType;
    super(options);
    _.bindAll(this, 'filterClicked');
  }
  filterClicked() {
    TypeScriptTinyIoC.raiseEvent(
      new bae.FilterEvent(
        this.model.get('filterValue'),
        this._filterType),
      bae.IIFilterEvent);
  }
}
```

We have added an `events` property to our `options` object, and registered a handler function for the `click` DOM event. Whenever someone clicks on a `FilterItemView`, Marionette will invoke the `filterClicked` function. We have also added a call to `_.bindAll` for this event, to ensure that the `this` variable is scoped to the class instance whenever the `filterClicked` function is called.

Remember that each instance of this `FilterItemView` has a corresponding `FilterValue` model available to it via the internal `model` property. So, within our `filterClicked` function, we are simply raising a new `FilterEvent`, using properties from the internal `model` variable.

Our event definition interfaces are as follows – again, please refer to the sample code for the matching `IInterfaceChecker` definitions:

```
export interface IFilterEvent {
  filterType: fm.FilterType;
  filterName: string;
}
export interface IFilterEvent_Handler {
  handle_FilterEvent(event: IFilterEvent);
}
```

We can now register handlers for these filter events elsewhere in our code. The logical place to put this event handler is on the `PageViewLayout` itself, as this class is responsible for rendering the board list. We will define our `handle_FilterEvent` function on the `PageViewLayout` as follows:

```
handle_FilterEvent(event: ev.IFilterEvent) {

  var mainCollection: bm.ManufacturerCollection =
    TypeScriptTinyIoC.resolve(bm.IManufacturerCollection);
  var filteredCollection;
  if (event.filterType == fm.FilterType.BoardType)
    filteredCollection = new bm.ManufacturerCollection(
      mainCollection.filterByBoardType(event.filterName));
  else if (event.filterType == fm.FilterType.Manufacturer)
    filteredCollection = new bm.ManufacturerCollection(
      mainCollection.filterByManufacturer(
        event.filterName));
  else if (event.filterType == fm.FilterType.None)
    filteredCollection = mainCollection;

  this._manufacturerView.collection = filteredCollection;
  this._manufacturerView.render();
}
```

This function starts by obtaining a reference to our "global" registered `ManufacturerCollection`. We then define a variable named `filteredCollection` to hold our filtered version of the main `ManufacturerCollection`. Based on the `FilterType` within the event itself, we call either `filterByBoardType`, or `filterByManufacturer`. If the event type is `FilterType.None`, we simply set the `filteredCollection` to the `mainCollection`, effectively clearing all filters.

The last part of this function sets the internal `collection` property of our main view (`this._manufacturerView`) to the resultant `filteredCollection`, and then calls `render`.

Our application is now responding to a click event on the `FilterItemView`, raising an event, and re-rendering the `ManufacturerView`, in order to apply the selected filter to our data for rendering.

Triggering a Detail view event

We have another click event, however, that we need to respond to. When a user clicks on a particular board, we need to fire an event that will slide the panels over, and show the board detail view.

Before we move onto the board detail view and how it is rendered, let's first hook up a click event on the `BoardView` class. To do so, we just need to specify a click event handler on the `options.events` parameters on the `BoardView` class, similar to our previous click event handler. We will also need to create an `onClicked` function, as follows:

```
export class BoardView
  extends Marionette.CompositeView<bm.BoardModel> {
  constructor(options?: any) {
    // existing code
    options.events = {
      "click": this.onClicked,
    };

    super(options);

    // existing code
    _.bindAll(this, 'onClicked');
  }

  onClicked() {
    this.$el.find('.board_inner_panel').flip({
      direction: 'lr',
      speed: 100,
      onEnd: () => {
        TypeScriptTinyIoC.raiseEvent(
          new bae.BoardSelectedEvent(this.model),
          bae.IIBoardSelectedEvent);
      }
    });
  }
}
```

The changes to this class are fairly minimal, we just set the `events` property on our options correctly, issue a call to `_.bindAll`, as we did in our `FilterItem` code, and then write an `onClicked` function. This `onClicked` function issues a call to `flip` as we saw in *Chapter 7, Modularization*, and then raises a new `BoardSelectedEvent`. Our `BoardSelectedEvent` interface and handler interfaces are as follows – again, please refer to the sample code for the matching `IInterfaceChecker` definitions:

```
export interface IBoardSelectEvent {
  selectedBoard: bm.BoardModel;
}
export interface IBoardSelectedEvent_Handler {
  handle_BoardSelectedEvent(event: IBoardSelectEvent);
}
```

The `BoardSelectedEvent` simply contains the entire `BoardModel` itself, in the `selectedBoard` property. With these event interfaces and classes in place, we can now register for a `BoardSelectedEvent` anywhere in our code.

Rendering the BoardDetailView

In this application, the logical place for handling this `BoardSelectedEvent` would be in the `PageViewLayout`, as it is responsible for cycling the carousel panels, and rendering the `BoardDetailView`. Let's update this class as follows:

```
export class PageViewLayout extends
  Marionette.LayoutView<Backbone.Model>
  implements ev.INotifyEvent_Handler,
  ev.IBoardSelectedEvent_Handler,
  ev.IFilterEvent_Handler
{
  // existing code
  constructor(options?: any) {
    // existing code
    _.bindAll(this, 'handle_NotifyEvent');
    _.bindAll(this, 'handle_BoardSelectedEvent');
    TypeScriptTinyIoC.registerHandler(this,
      ev.IINotifyEvent_Handler, ev.IINotifyEvent);
    TypeScriptTinyIoC.registerHandler(this,
      ev.IIBoardSelectedEvent_Handler,
      ev.IIBoardSelectedEvent);
  }
  handle_BoardSelectedEvent(event: ev.IBoardSelectEvent) {
    var boardDetailView = new bdv.BoardDetailView(
```



```
        { model: event.selectedBoard });  
        boardDetailView.render();  
    }  
}
```

Here, we have updated our `PageViewLayout` class to implement the `IBoardSelectedEvent_Handler` interface, and registered it with `TypeScriptTinyIoC`. We are responding to the `BoardSelectedEvent` by creating a new `BoardDetailView` class, using the full `BoardModel` included in the event, and then calling `render`. Our `BoardDetailView` class is as follows:

```
export class BoardDetailView  
    extends Marionette.CompositeView<bm.BoardSize> {  
    constructor(options?: any) {  
        if (!options)  
            options = {};  
  
        options.el = "#board_detail_view";  
        var snippetService: ISnippetService =  
            TypeScriptTinyIoC.resolve(IISnippetService);  
        options.template = _.template(  
            snippetService.retrieveSnippet(  
                SnippetKey.BOARD_DETAIL_VIEW_SNIPPET));  
  
        super(options);  
  
        this.collection = <any>(  
            new Backbone.Collection(this.model.get('sizes')));  
        this.childView = mv.BoardSizeView;  
        this.childViewContainer = 'tbody';  
  
        var snippetService: ISnippetService =  
            TypeScriptTinyIoC.resolve(IISnippetService);  
        this.childViewOptions = {  
            template: _.template(  
                snippetService.retrieveSnippet(  
                    SnippetKey.BOARD_SIZE_VIEW_SNIPPET)),  
            tagName: 'tr'  
        };  
    }  
}
```

The `BoardDetailView` class is very similar to our `BoardView`, but it uses the `"#board_detail_view"` element for the `options.el` property – which is our corresponding DOM element. Our snippet has the `BOARD_DETAIL_VIEW_SNIPPET` key. We then create a `Backbone.Collection` out of the `sizes` property, and set the `childView` to the `BoardSize` view class template, in the same way that we did earlier for the `BoardView`.

Our `childViewContainer`, however, now targets the `<tbody></tbody>` tag to render children into. We are also passing the template from the `BOARD_SIZE_VIEW_SNIPPET` to the child `BoardSize` view, and setting the `tagName` to `'tr'`. Remember how we moved the configuration of the child `BoardSize` views up one level in our `BoardView`? Well, we are doing the same thing here.

Please refer to the sample code for a full listing of the `BoardDetailViewSnippet.html`, and the `BoardSizeViewSnippet.html`.

The State Design Pattern

Our last task for this application is to control the various screen elements as users interact with our application. As a user navigates the application, we need to move from carousel panel 1 to carousel panel 2, and update screen elements, such as showing and hiding the left-hand side filter panel. In a large web application, there may be many screen elements, many different transitions, and things such as pop ups or masks that say **"loading..."** while our application fetches data from backend services. Keeping track of all of these elements becomes a difficult and time-consuming task, often leaving large swathes of if-else or switch statements in many different areas of our code, leading to a lot of direct DOM manipulation spaghetti.

The State Design Pattern is a design pattern that can simplify our application code, so that code that manipulates these various DOM elements can reside in one place. The State Design Pattern defines a set of states that the application could be in, and provides an easy mechanism to transition between these states, control visual screen elements, and handle animations.

Problem space

As an example of what we are trying to achieve, consider the following business rules:

- When a user first logs into the BoardSales application on a desktop, the left-hand filter panel should be visible.
- If the user is using a mobile device, the left-hand filter panel should not be visible when a user first logs in. This is done to save on screen real estate.

- If the filter panel is visible, then the expand icon should switch to a left-hand arrow (<) to allow the user to hide it.
- If the filter panel is not visible, then the expand icon should be a right-hand arrow (>) to allow the user to show it.
- If a user expands the filter panel, and then switches to a board detail view and back again, then the filter panel should remain expanded.
- If a user hides the filter panel, and then switches to a board detail view and back again, then the filter panel should remain hidden.

On top of these business rules, we have an outstanding bug that has been reported for users on a Firefox browser (you can test this behavior using the demo HTML page):

When clicking on a board in the board list view, with the filter panel open, the carousel panel does not behave correctly. The carousel first cycles across to the board detail view, and then closes the filter panel. This transition is inconsistent with other browsers, where the filter panel is cycled along with the board list at the same time.

This bug therefore adds another business requirement to our list:

- For users on a Firefox browser, please hide the filter panel first, before cycling the carousel to the board detail view.

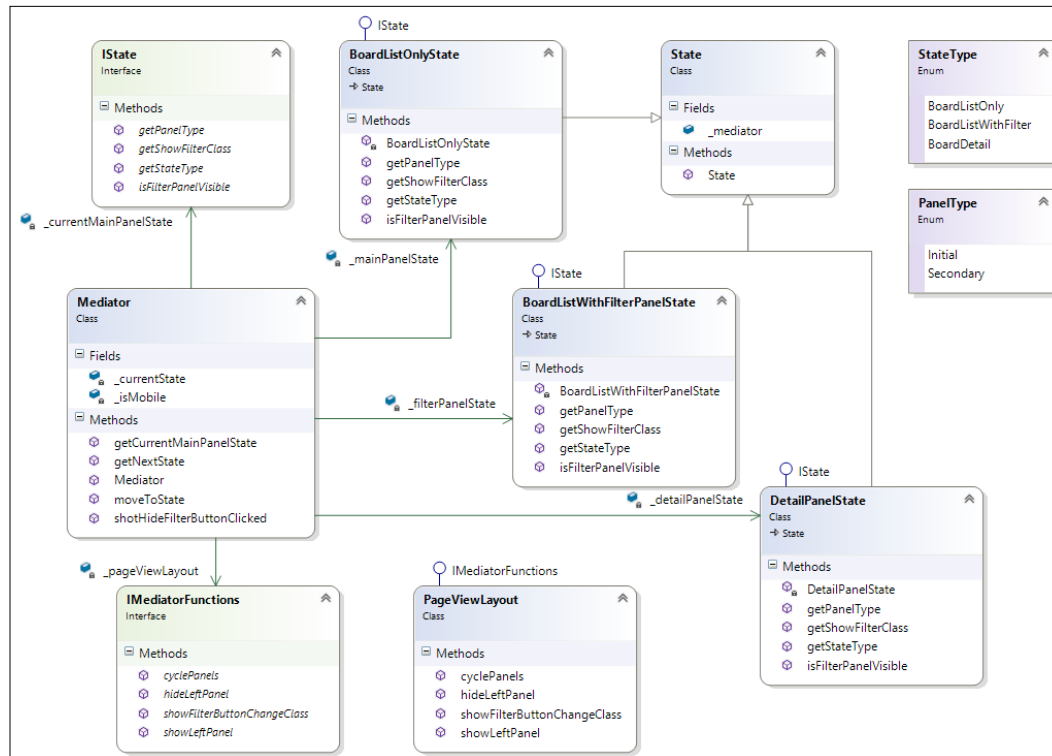
The State Design Pattern uses a set of very similar classes, each representing a particular application state. Each of these state classes are derived from the same base class. When we want our application to change to a different state, we simply switch to the object that represents the state that we are interested in.

For example, our application really has only three states. We have a state where the board list and filter panels are both visible. We have another state where only the board list is visible, and our third state is where the board detail panel is visible. Depending on which state we are in, we should be either on `carousel_panel_1`, or on `carousel_panel_2`. Also, the icon that is used in conjunction with the filter panel needs to switch from a left-hand chevron < to a right-hand chevron >, depending on the application state.

The State Design Pattern also has the concept of a Mediator class, which will keep track of the current state, and contain the logic of how to switch between each of these states.

State class diagram

Consider the following class diagram for the State and Mediator Design Pattern:



State and Mediator pattern class diagram

We start with an enum named `StateType` that lists our three application states, and second enum named `PanelType` to indicate which carousel panel each of these states are on. We then define an interface named `IState` that each of these states must implement. To hold properties common to each state, we have also defined a base state class, from which all states will derive. Our implementation of these enums, the `IState` interface, and the base `State` class as follows:

```
export enum StateType {
  BoardListOnly,
  BoardListWithFilter,
  BoardDetail,
}
```

```
export enum PanelType { Initial, Secondary }
export interface IState {
  getPanelType(): PanelType;
  getStateType(): StateType;
  getShowFilterClass(): string;
  isFilterPanelVisible(): boolean;
}
export class State {
  private _mediator: sm.Mediator;
  constructor(mediator: sm.Mediator) {
    this._mediator = mediator;
  }
}
```

Our `StateType` enum has defined each state that we will be using. Our application, therefore, is either in `BoardListOnly` state, `BoardListWithFilter` state, or `BoardDetail` state. Our second enum, named `PanelType`, is used to indicate which of the carousel panels we are currently on, either the `Initial` panel (`carousel_panel_1`), or the `Secondary` panel (`carousel_panel_2`).

We then define an `IState` interface that all state objects must implement. This interface allows us to query each state, and determine four important pieces of information. The `getPanelType` function tells us what panel we should be currently viewing, and the `getStateType` function returns the `StateType` enum value. The `getShowFilterClass` function will return a string that is used to apply a CSS class to the show / hide filter button, and the `isFilterPanelVisible` function returns a boolean to indicate whether or not the filter panel is visible.

Each state needs a reference to the `Mediator` class, so we have created a base `State` class with a `constructor` function, from which each of our `State` objects can be derived from.

Concrete State classes

Let's now create concrete classes for each of these states. The first state that our application can be in, is when we are viewing the board list, and the filter panel is hidden:

```
export class BoardListOnlyState
  extends ss.State
  implements ss.IState {
  constructor(mediator: sm.Mediator) {
    super(mediator);
  }
}
```

```

    getPanelType(): ss.PanelType {
        return ss.PanelType.Initial;
    }
    getShowFilterClass() {
        return "glyphicon-chevron-right";
    }
    isFilterPanelVisible(): boolean {
        return false;
    }
    getStateType(): ss.StateType {
        return ss.StateType.BoardListOnly;
    }
}

```

Our `BoardListOnlyState` class extends the `State` class that we defined earlier, and implements the `IState` interface. In this `BoardListOnly` state, we should be on the `Initial` carousel panel, the class to be used for the show / hide filter panel button should be a `glyphicon-chevron-right` [>], and the left-hand side filter panel should NOT be visible.

The next state that our application could be in, is when the board list is showing, and we also have the filter panel visible:

```

export class BoardListWithFilterPanelState
    extends ss.State
    implements ss.IState {
    constructor(mediator: sm.Mediator) {
        super(mediator);
    }
    getPanelType(): ss.PanelType {
        return ss.PanelType.Initial;
    }
    getShowFilterClass() {
        return "glyphicon-chevron-left";
    }
    isFilterPanelVisible(): boolean {
        return true;
    }
    getStateType(): ss.StateType {
        return ss.StateType.BoardListWithFilter;
    }
}

```

In the `BoardListWithFilterPanel` state, our carousel panel is again the `Initial` panel, but our class for the show / hide filter panel button is now a `glyphicon-chevron-left (<)`. Our filter panel is also visible.

The last state we need to define for our application, is when we have cycled over to `carousel_panel_2`, and are viewing the board detail screen:

```
export class DetailPanelState
  extends ss.State
  implements ss.IState {
  constructor(mediator: sm.Mediator) {
    super(mediator);
  }
  getPanelType(): ss.PanelType {
    return ss.PanelType.Secondary;
  }
  getShowFilterClass() {
    return "";
  }
  isFilterPanelVisible(): boolean {
    return false;
  }
  getStateType(): ss.StateType {
    return ss.StateType.BoardDetail;
  }
}
```

In the `DetailPanel` state, we are on the `Secondary` carousel panel, we do not need a class for the show / hide filter panel button (as the panel has moved off the screen), and the filter panel itself is NOT visible.

Note that in the sample application source code, you will find a series of unit tests that will test each of these properties. For the purposes of brevity, we will not list them here.

The Mediator class

In object-oriented patterns, a Mediator is used to encapsulate the logic of how a set of objects interacts. In our case, we have a set of states that define what visual elements should be shown. There is also a need to define how these various elements transition according to the movement between these states.

We will, therefore, define a `Mediator` class to encapsulate all of this transition logic, and co-ordinate the changes to our visual elements, based on movements between states. In order for our `Mediator` class to interact with the UI, we will define a set of four functions that must be implemented by any class using this `Mediator`:

```
export interface IMediatorFunctions {
  showLeftPanel();
  hideLeftPanel();
  cyclePanels(forwardOrNext: string);
  showFilterButtonChangeClass(
    fromClass: string, toClass: string
  );
}
```

Our `IMediatorFunctions` interface has four functions. The `showLeftPanel` function will show our filter panel. The `hideLeftPanel` function will hide the filter panel. The `cyclePanels` function will be invoked with either a 'prev' string, or a 'next' string to cycle the carousel panel from `carousel_panel_1` to `carousel_panel_2`. The `showFilterButtonChangeClass` will be invoked with two arguments—a `fromClass` string that is a CSS class, and a `toClass` string that is another CSS class. This function will just remove the `fromClass` CSS class from the DOM element, and then add the `toClass` CSS class to the DOM element. In this way, we can change the icon used for our show / hide filter button from a chevron-right (>) to a chevron-left (<).

We can now look at the internal logic of the `Mediator` class itself, starting with a set of private variables and the constructor:

```
export class Mediator {
  private _currentState: ss.IState;
  private _currentMainPanelState: ss.IState;
  private _pageViewLayout: IMediatorFunctions;
  private _isMobile: boolean;

  private _mainPanelState: as.BoardListOnlyState;
  private _detailPanelState: as.DetailPanelState;
  private _filterPanelState: as.BoardListWithFilterPanelState;

  constructor(pageViewLayout: IMediatorFunctions,
    isMobile: boolean) {
    this._pageViewLayout = pageViewLayout;
    this._isMobile = isMobile;

    this._mainPanelState = new as.BoardListOnlyState(this);
```



```
        this._detailPanelState = new as.DetailPanelState(this);
        this._filterPanelState =
            new as.BoardListWithFilterPanelState(this);

        if (this._isMobile)
            this._currentState = this._mainPanelState;
        else
            this._currentState = this._filterPanelState;
        this._currentMainPanelState = this._currentState;
    }
}
```

Our Mediator class has a number of private variables. The `_currentState` variable is used to hold an instance of one of our State classes, and represents the current state of the UI. This `_currentState` variable can hold any one of our three states. The `_currentMainPanelState` variable again holds one of our State classes, but represents the current state of the main panel. This `_currentMainPanelState` will only hold either a `BoardListOnlyState`, or a `BoardListWithFilterPanelState`.

The `_pageViewLayout` variable will hold an instance of the class that implements our `IMediatorFunctions` interface, and we will apply state changes to the UI through this variable. For those of you familiar with the MVP pattern, the Mediator class is acting as a Presenter, and the `_pageViewLayout` variable is acting as the View.

The `_isMobile` variable just holds a boolean value indicating whether or not we are on a mobile device. We will set this variable a little later.

We then have three private variables that will hold instances of our three states—`BoardListOnlyState`, `DetailPanelState`, and `BoardListWithFilterPanelState`.

Our constructor simply sets up these private variables, and then instantiates an instance of each of our state classes, assigning them to the correct internal variable.

Note the code near the bottom of the constructor. This is the implementation of one of our business rules. If the application is being viewed on a mobile device, then the filter panel should NOT be visible by default. We are, therefore, setting the value of the `_currentState` variable to one of the initial states, based on our `isMobile` flag. To round out our constructor function, we also set the initial value of the `_currentMainPanelState` variable to the `_currentState`.

Our next Mediator function, `getNextState`, simply returns one of our private State variables, using a `StateType` enum as input:

```
private getNextState(stateType: ss.StateType): ss.IState {
    var nextState: ss.IState;
    switch (stateType) {
```

```

        case ss.StateType.BoardDetail:
            nextState = this._detailPanelState;
            break;
        case ss.StateType.BoardListOnly:
            nextState = this._mainPanelState;
            break;
        case ss.StateType.BoardListWithFilter:
            nextState = this._filterPanelState;
    }
    return nextState;
}

```

This is essentially a mini factory method, and will return the correct internal `State` object, based on the value of the `StateType` argument.

Moving to a new State

The main body of logic that controls how the UI needs to be updated, based on the movement between states, is implemented in the `moveToState` function, as follows:

```

public moveToState(stateType: ss.StateType) {
    var previousState = this._currentState;
    var nextState = this.getNextState(stateType);

    if (previousState.getPanelType() == ss.PanelType.Initial &&
        nextState.getPanelType() == ss.PanelType.Secondary) {
        this._pageViewLayout.hideLeftPanel();
        this._pageViewLayout.cyclePanels('next');
    }

    if (previousState.getPanelType() == ss.PanelType.Secondary &&
        nextState.getPanelType() == ss.PanelType.Initial) {
        this._pageViewLayout.cyclePanels('prev');
    }

    this._pageViewLayout.showFilterButtonChangeClass(
        previousState.getShowFilterClass(),
        nextState.getShowFilterClass()
    );

    if (nextState.isFilterPanelVisible())
        this._pageViewLayout.showLeftPanel();
    else

```

```
        this._pageViewLayout.hideLeftPanel();

        this._currentState = nextState;
        if (this._currentState.getStateType() ==
            ss.StateType.BoardListOnly
            || this._currentState.getStateType() ==
            ss.StateType.BoardListWithFilter)
            this._currentMainPanelState = this._currentState;
    }
```

This function will be called whenever we want to move from one state to another. The first thing this function does, is to set up two variables: `previousState` and `nextState`. The `previousState` variable is actually our current state object, and the `nextState` variable is a `State` object for the state that we are moving to.

We can now compare the `previousState` variable with the `nextState` variable and make some decisions.

The logic for our first if statement goes something like this: if we are moving from an `Initial` panel type to a `Secondary` panel, then call the relevant functions on the UI to hide the left panel, and initiate a carousel cycle to 'next'. This logic will fix the Firefox bug that we were notified of earlier.

The logic for our second if statement is the opposite of the first one: if we are moving from a `Secondary` panel to an `Initial` panel, then initiate a carousel cycle with 'prev'.

The next step in our logic applies the class for the show / hide filter button to the UI, by calling the `showFilterButtonChangeClass` function on the UI, passing in the CSS class name from the `previousState`, and the CSS class name from the `nextState` as arguments. Remember that this will remove the CSS class from `previousState`, and then add the CSS class from `nextState` to the show / hide filter button CSS.

Our next logical step checks whether the filter panel should be shown or hidden, and calls the corresponding function on our `_pageViewLayout`.

As we are now done with our state change logic, and can set the value of the `_currentState` variable to now hold our `nextState`.

The last piece of logic just checks to see whether we are currently in `BoardListOnly` or `BoardListWithFilter` state, and if so, stores the current state in the `_currentMainPanelState` variable. This logic will form part of the business rules that we have been given, to ensure that when we switch from our main panel to our detail panel and back again, the status of the filter panel is maintained correctly.

We have two more functions in our `Mediator` class to discuss, which are as follows:

```
public showHideFilterButtonClicked() {
    switch (this._currentState.getStateType()) {
        case ss.StateType.BoardListWithFilter:
            this.moveToState(ss.StateType.BoardListOnly);
            break;
        case ss.StateType.BoardListOnly:
            this.moveToState(ss.StateType.BoardListWithFilter);
            break;
    }
}

public getCurrentMainPanelState(): ss.IState {
    return this._currentMainPanelState;
}
```

The first function, called `showHideFilterButtonClicked` is actually the function that will need to be called when we click on the show / hide filter button in our application. Depending on whether the filter panel is open or closed, the behavior of this button will be slightly different. The only object that knows what to do, depending on what state the application is in, is the `Mediator` class itself. So, we are deferring the decision-making on what to do when the button is clicked, to the `Mediator` class.

The implementation of the `showHideFilterButtonClicked` function just checks what our current state is, and then calls a `moveToState` with the correct `nextState` as the parameter.



When you are building a large-scale application, there may be many different buttons or screen elements that change slightly depending on what state your application is in. Deferring the decision-making logic to a `Mediator` class provides a simple and elegant way of managing all of your screen elements. This business logic is captured in one place, and can also be tested thoroughly. Be sure to check the sample code for a full suite of tests surrounding the `Mediator` class.

Our final function, `getCurrentMainPanelState`, simply returns the last known state of our main panel, and will be used to implement the business logic for remembering whether the filter panel is open or closed.

Implementing the IMediatorFunctions interface

When the `Mediator` class needs to trigger changes to the UI, it calls functions on the `IMediatorFunctions` interface, as we have seen previously. Our application, therefore, must implement this `IMediatorFunctions` interface somewhere. As the `PageViewLayout` class holds references to each of the UI elements we need to change, the logical place to implement this interface is on the `PageViewLayout` class itself, as follows:

```
export class PageViewLayout extends
  Marionette.LayoutView<Backbone.Model>
  implements ev.INotifyEvent_Handler,
  ev.IBoardSelectedEvent_Handler,
  ev.IFilterEvent_Handler,
  sm.IMediatorFunctions
{
  private _mediator: sm.Mediator;
  constructor(options?: any) {
    // existing code
    options.events = {
      "click #show_filter_button":
        this.showHideFilterButtonClicked
    };
    // existing code
    var isMobile = $('html').hasClass('mobile');
    this._mediator = new sm.Mediator(this, isMobile);
    // existing code
  }
  // existing functions
  showLeftPanel() {
    $('#content_panel_left')
      .removeClass('sidebar_panel_push_to_left');
    $('#content_panel_main')
      .removeClass('main_panel_push_to_left');
  }
  hideLeftPanel() {
    $('#content_panel_left')
      .addClass('sidebar_panel_push_to_left');
    $('#content_panel_main')
      .addClass('main_panel_push_to_left');
  }
  cyclePanels(forwardOrNext: string) {
```

```

        $('#carousel-main-container').carousel(forwardOrNext);
    }
    showFilterButtonChangeClass(
        fromClass: string, toClass: string) {
        $('#show_filter_button')
            .removeClass(fromClass).addClass(toClass);
    }
    showHideFilterButtonClicked() {
        this._mediator.showHideFilterButtonClicked();
    }
    // existing functions
}

```

We have updated our `PageViewLayout` class to implement all of the functions in the `IMediatorFunctions` interface. We have also included a private variable named `_mediator` to hold an instance of the `Mediator` class, and set this up in our constructor.

As with our other views that need to respond to click events, we have set up an `options.events` object to tie a DOM `click` event on the `#show_filter_button` DOM element (which is our show / hide button), to the `showHideFilterButtonClicked` function.



We are using jQuery to check whether the main HTML element in our page has a class named `mobile`. This class will be set by the `head.js` utility script that we included in our `index.html` page at the beginning of this chapter. In this way, we are able to determine whether our application is being used on a mobile or desktop device.

The `showLeftPanel` and `hideLeftPanel` functions just include the jQuery snippets to apply or remove the relevant classes, in order to slide the filter panel in or out.

The `cyclePanels` function calls our Bootstrap carousel function with either a `'next'` or `'prev'` parameter, as we did in our demo HTML page.

The `showFilterButtonChangeClass` simply removes the `fromClass` CSS style from our `show_filter_button` DOM element, and then adds the new `toClass` CSS style. Removing and adding these CSS classes will switch the button displayed from a left chevron (`<`) to a right chevron (`>`), or vice versa.

When a user clicks on the `#show_filter_button` DOM element, our `showHideFilterButtonClicked` method will be invoked. As discussed earlier, we are forwarding this call to the `Mediator` instance, so that the `Mediator` logic can make the decision as to what to do when the button is clicked.

Triggering State changes

To finish off our State and Mediator Design Pattern, we will now just need to call the Mediator functions in the right places, in order to trigger the logic to move to a different state.

The first place we call the `moveToState` function is in our `handle_NotifyEvent`, when our `ManufacturerDataLoaded` event is triggered. This event only ever occurs once in our application, and that is after the `ManufacturerCollection` has been successfully loaded. We already have an event handler for this in our `PageViewLayout` class, so let's update this function as follows:

```
handle_NotifyEvent(event: ev.INotifyEvent) {
    if (event.eventType == ev.EventType.ManufacturerDataLoaded) {
        // existing code
        this._manufacturerView =
            new mv.ManufacturerCollectionView();
        this._manufacturerView.render();

        this._mediator.moveToState(
            this._mediator
                .getCurrentMainPanelState().getStateType()
        );
    }
    if (event.eventType == ev.EventType.BoardDetailBackClicked) {
        this._mediator.moveToState(
            this._mediator.getCurrentMainPanelState()
                .getStateType()
        );
    }
}
```

Our first `if` statement checks for the `ManufacturerDataLoaded` event type, and then creates a new `ManufacturerCollectionView` and calls its `render` function, as we have seen previously. We then call the `moveToState` function, passing in the Mediator's `currentMainPanelState` as an argument. Remember how we set the initial main panel state in the Mediator's constructor, based on whether or not the browser was on a mobile device? This call to `moveToState` will use that initial state as a parameter, essentially starting the application in the correct state.

Our second `if` statement will trigger a `moveToState` when the user is in the `BoardDetail` screen, and clicks on the back button on the header panel. This logic again uses the `currentMainPanelState` to restore our board list to the correct state, according to our business rules.

The other function within the `PageLayoutView` that will trigger a call to `moveToState`, is our handler for a `BoardSelectedEvent`:

```
handle_BoardSelectedEvent(event: ev.IBoardSelectEvent) {
  var boardDetailView = new bdv.BoardDetailView(
    { model: event.selectedBoard });
  boardDetailView.render();

  this._mediator.moveToState(ss.StateType.BoardDetail);
}
```

Whenever a user clicks on a board in the board list, a `BoardSelectedEvent` is triggered, and we render the `BoardDetailView`. This `BoardDetailView`, however, is on the second carousel panel, so we will need to move to the `BoardDetail` state as part of this event handler.

Lastly, we will need to trigger the `moveToState` function when the user is in a `BoardDetailView`, and clicks on the back button. To implement this, we will need to raise a `NotifyEvent`, with the `eventType` set to `BoardDetailBackClicked`, from our `BoardDetailView`, as follows:

```
export class BoardDetailView
  extends Marionette.CompositeView<bm.BoardSize> {
  constructor(options?: any) {
    // existing code
    options.events = {
      "click #prev_button": this.onPrev
    };
    super(options);
    // existing code
  }

  onPrev() {
    TypeScriptTinyIoC.raiseEvent(
      new
        bae.NotifyEvent(bae.EventType.BoardDetailBackClicked),
      bae.IINotifyEvent);
  }
}
```

Here, we have tied the `onPrev` function to the DOM `click` event on the `#prev_button` element. Once a click is triggered, we simply need to raise a new `NotifyEvent`, with the `eventType` set to `BoardDetailBackClicked`, in order to trigger a `moveToState` function call.

With our State and Mediator Design Pattern classes in place, our sample application is now complete.

Summary

In this chapter, we built a full TypeScript single-page application from the ground up. We started with an initial idea of how the application would be designed, and how we wanted the pages to transition. We then built a pure HTML demo page using out-of-the-box Bootstrap elements, and sprinkled a little JavaScript magic to create a full demo page. We applied various styles to the HTML, previewed it in Brackets, and tweaked the look and feel until we were happy.

Our next major step was to understand, and work with, the data structures that we would need within our application. We wrote Jasmine unit tests and integration tests to solidify our Backbone models and collections, and wrote the filtering functions that we needed.

We then built up a set of Marionette views, and split up our demo HTML into snippets for each of these views to use. We tied the views to our collections and models, and used interfaces to work with data providers. Our application then started to come together by working with real server-side data.

Finally, we discussed page transition strategies, and implemented a State and Mediator Design Pattern to implement our required business logic.

Hopefully, you have enjoyed the journey of building an application from the ground up—from concept to visualization, and then through implementation and testing. We have finally arrived at an industrial strength, enterprise ready, TypeScript single-page Marionette application.

Index

A

acceptance tests

- about 158
- versus integration tests 158
- versus unit tests 158

AMD

- about 197
- application, rendering 226-228
- Backbone 207
- Backbone.Collections, using 215-218
- Backbone views 219, 220
- collection, rendering 223, 224
- collections 207
- jQuery plugins, using 229-232
- model, creating 208, 209
- models 207
- Require config errors, fixing 213
- require.config file 210-213
- Text plugin, using 221, 222
- using 206
- views 207

Angular

- \$scope argument 147, 148
- about 144
- classes 146-148
- inheritance 149
- TypeScript compatibility 148

Angular 2.0 150

anonymous function 48, 49

any type 42

arrays 41

assembly 243

asynchronous functions

- chaining 202-205

Asynchronous Module Loading. *See* AMD

B

Backbone

- about 139, 207
- ECMAScript 5, using 143
- generic syntax, using 143
- inheritance, using 140, 141
- interfaces, using 142
- TypeScript compatibility 144
- views 219, 220

basic types

- syntax 37
- TypeScript 37

black box tests 157

Board Sales 272

Bootstrap

- about 272
- carousel panel 1 element 275
- carousel panel 2 276
- elements 274, 275
- filter options 275
- installing 273
- show / hide panel button 275
- using 274, 275

Bower package manager 185

Brackets

- about 24
- debugging, in Chrome 33
- Grunt, using 30-32
- installing 24, 25
- live preview, using 27, 28
- npm (Node package manager) 31
- project, creating 26, 27
- TypeScript, compiling 30
- TypeScript file, creating 28, 29
- URL 24

build server, selecting
Jenkins 160
TeamCity 161
Team Foundation Server (TFS) 160

C

callback function 53, 54

CI

about 158
benefits 159
build server, selecting 160
using 159

class

about 65
constructor overloads 125
constructors 66
creating, in Angular 146-148
creating, in ExtJs 150, 152
declaration file syntax 125
functions 67-69, 126
interface function definitions 70, 71
JavaScript syntax 124
namespaces 125
optional properties 128
properties 126

class constructor overloads

about 125
declaration file syntax 125
JavaScript syntax 125

class modifiers

about 82
class property accessors 84
constructor access modifiers 83, 84
static functions 85
static properties 86, 87

class namespaces

about 125
declaration file syntax 125
JavaScript syntax 125

closures

generating, with TypeScript class 10

collection

about 207
board types, finding 287, 288

manufacturer names, finding 286
rendering 223, 224
traversing 284, 285

CommonJs

about 198
asynchronous functions, chaining 202-205
Node module, creating 200, 201
Node module, using 201
Node, setting up in Visual Studio 198-200

config errors, Require

404 errors 214, 215
fixing 213
incorrect dependencies 214

constructor function 67

Continuous Integration. *See* CI

Curl

URL 198

D

data structure

about 277, 278
collection, filtering 288-290
collection, traversing 284
data interfaces 278-281
integration tests 282, 283

data types, TypeScript

arrays 41
enum 41, 44

declaration file

about 107
function overrides 120, 121
installing 135
interfaces 117-119
module keyword 115-117
rounding out 122
syntax reference 123
writing 112-114

declaration syntax reference

about 123
class constructor overloads 125
classes 124
class functions 126
class namespaces 125
class properties 126

- function overrides 123
- function signatures 127
- functions, merging with module
 - definition 128
- global functions 127
- nested namespaces 124
- optional properties 128
- static functions 126
- static properties 126
- default parameters 50, 51**
- definition files**
 - downloading 132, 133
 - installing 138
- DefinitelyTyped repository**
 - about 8, 131
 - URL 8, 131
- Dependency Injection (DI)**
 - about 144, 243
 - versus Service Location 244, 245
- Dependency Inversion principle 237**
- Dependency resolution**
 - about 243
 - Dependency Injection 243
 - Service Location 243
- design pattern 65, 66**
- Domain Event Pattern**
 - about 255, 256
 - error notifications, displaying 267-269
 - event, firing 262, 263
 - event handler, registering 264-267
 - handler interface 257-259
 - message interface 257-259
 - multiple event handlers 259-262
 - problem space 256
 - reference link 256
- DOM events, Marionette**
 - about 310, 311
 - BoardDetailView, rendering 313-315
 - Detail view event, triggering 312, 313
- done() function**
 - using 173
- duck-typing method 40, 41**

E

ECMAScript 3, 4

ECMAScript 5

- using, with Backbone 143

Embedded JavaScript (EJS) 108

encapsulation 8-10

enum

- about 43-46

- const enums 46

error notifications

- displaying 267-269

event

- event handler, registering 264-267

- firing 262, 263

explicit casting 42, 43

Extension Manager

- using 134, 135

ExtJs

- about 150

- classes, creating 150-152

- type casting, using 152, 153

- TypeScript compiler 153

- URL, for documentation 150

F

Factory class

- about 79, 80

- using 80, 81

Factory Design Pattern

- about 63, 77

- business requirements 77

- Factory class 79, 80

- Factory class, using 80

- IPerson interface 78

- Person base class 78

- returned objects 79

- tasks 77, 78

function definition

- merging, with module definition 128

function overrides

- about 123

- declaration file syntax 124

- JavaScript syntax 124

functions

- about 47
- anonymous functions 48
- arguments variable 51-53
- callbacks 53-58
- default parameters 50, 51
- optional parameters 49, 50
- overloads 59, 60
- scope 56-58
- signatures 54, 55
- union types 60, 61

function signatures

- about 54, 55, 127
- declaration file syntax 127
- JavaScript syntax 127

G

Gang of Four (GoF) 235

generics

- about 87
- classes, instantiating 88, 89
- interfaces 94, 95
- syntax 87, 88
- type T, constraining 92
- type T, using 90, 91
- used, for checking interface 102-105

generic syntax

- using, with Backbone 143

getProperties function 74

global functions

- declaration file syntax 127
- JavaScript syntax 127

global variables 108, 109

Grunt

- about 30
- URL 30

H

Handlebars

- URL 220

handler interface 257-259

HTML

- JavaScript code blocks, using 110

I

IMediatorFunctions interface

- implementing 326, 327

Immediately Invoked Function Expression (IIFE) 76

inferred typing 39

inheritance

- about 71, 149
- Angular, versus Backbone 149
- class inheritance 72
- constructor, overloading with super keyword 73, 74
- function, overloading with super keyword 73, 74
- interface inheritance 71, 72
- JavaScript closures 75, 76
- using, with Backbone 140, 141

installation

- declaration files 135
- definition files 138

integration tests

- about 157, 182
- detailed test results 185
- page elements, searching 190, 191
- simulating 182-184
- test results, logging 186-190
- versus acceptance tests 157
- versus unit tests 157

IntelliSense 5

interface

- about 64, 65
- programming 236
- using, with Backbone 142

Interface Segregation principle 237

Inversion of Control (IOC) 105

J

Jasmine

- about 161
- asynchronous tests 171, 172
- data-driven tests 167
- DOM events 175, 176

- done() function, using 173
- fixtures 174, 175
- matchers 165 page elements, working with 192-194
- running, ways 176
- SpecRunner.html file 163, 164
- spies, using 168-170
- spies, using as fakes 170, 171
- teardown 166
- test 162
- test startup 166
- URL 161
- Jasmine, runners**
 - about 176
 - Karma 178, 179
 - Protractor 180
 - Testem 177, 178
- JavaScript**
 - about 1, 36
 - basic types 36
 - framework, selecting 138, 139
 - functions 47
 - variables, defining 36
- JavaScript code blocks**
 - structured data 111, 112
 - using, in HTML 110
- Jenkins 160**
- jQuery plugins**
 - using 229-232

K

- Karma 178, 179

L

- Liskov Substitution principle 237

M

- Marionette**
 - about 271
 - DOM events 310
 - views 297
- Marionette application**
 - building 290-293
 - main collection, loading 294-297
- Marionette views**
 - about 297, 298
 - BoardSizeView class 302
 - BoardView class 300, 301
 - FilterCollection class 305-308
 - filtering 308, 309
 - filtering, with IFilterProvider interface 304, 305
 - ManufacturerCollectionView class 298, 299
 - ManufacturerView class 300
- MaxUnit**
 - URL 161
- message interface 257-259**
- model**
 - about 207
 - creating 208
- Model View Presenter (MVP) 155**
- Model View Something (MV*) 155**
- Model View ViewModel (MVVM) 155**
- Model View Whatever (MVW) 155**
- modularization 197**
- module**
 - merging 122, 123
- multiple event handlers 259-262**

N

- nested function 56**
- nested namespaces**
 - declaration file syntax 124
 - JavaScript syntax 124
- Node module**
 - creating 200, 201
 - using 201
- NuGet**
 - about 134
 - declaration files, installing 135
 - Extension Manager, using 134, 135
 - Package Manager Console, using 136
 - using 134

O

Open Closed principle 237

P

Package Manager Console

package names, searching 136

packages, installing 136

specific version, installing 136

using 136

packages, TSD

querying 137

page layout

about 272

board detail panel 273

board listing panel 273

Bootstrap, installing 273

Bootstrap, using 274

filter panel 273

footer panel 273

viewing panel 273

print function 66

private accessors 10, 11

properties, class

declaration file syntax 126

JavaScript syntax 126

Protractor

about 180

Selenium, using 180-182

URL 180

public accessors 10, 11

Q

QUnit

URL 161

R

reflection 236

Representational State Transfer (REST) 157

require.config file 210-213

runtime

interface, checking with generics 102-105

object, checking for function 101

reflection-like capabilities implementation,
issues 100

reflection process 98, 99

type, checking 96-98

S

Scripts directory 135

Selenium IDE

URL 191

Separation of Concerns principle 197

Service Location

about 243

versus Dependency Injection 244, 245

Service Locator

building 238

classes, registering against named

interfaces 247-250

implementing 245

named interfaces, writing 246

problem space 238-240

service, creating 240-242

testing 254

using 250-253

Single Responsibility principle 236

SOLID principles

about 236

Dependency Inversion 237

Interface Segregation 237

Liskov Substitution 237

Open Closed 237

Single Responsibility 236

State Design Pattern

about 315

business rules 315

changes, triggering 328, 329

class diagram 317, 318

concrete classes 318-320

IMediatorFunctions interface,

implementing 326, 327

Mediator class 320-323

Mediator class, moveToState

function 323-325

static functions

about 126

declaration file syntax 126

JavaScript syntax 126

static properties 126

T

TeamCity 161

Team Foundation Server (TFS) 160

Test Driven Development (TDD)

about 156

steps 156

Testem

about 177, 178

URL 177

Text plugin

using 221, 222

third-party libraries

JavaScript framework, selecting 138, 139

using 138

TSD

definition files, installing 138

packages, querying 137

URL 137

using 137

wildcards, using 138

type casting

using 152, 153

TypeScript

about 2, 3, 37

any type 42

benefits 4

compatibility, with Angular 148

compatibility, with Backbone 144

compiler 153

data types 41

DefinitelyTyped 8

duck-typing 40, 41

ECMAScript 3

encapsulation 8

explicit casting 42, 43

inferred typing 39

JavaScript libraries, defining 6, 7

private accessors 10

public accessors 10

strong typing 5

syntactic sugar 6

syntax 37-39

TypeScript Definition Manager. *See* TSD

TypeScript IDEs

about 12

Brackets 24

debugging, in Visual Studio 15, 16

Visual Studio 2013 12

WebStorm 17

U

union types

aliases 61, 62

guards 61

unit tests

about 157

versus acceptance tests 157

versus integration tests 157

User Acceptance Testing (UAT) 185

V

Visual Studio

Node, setting up TypeScript,

debugging 15, 16, 198-200

Visual Studio 2013

about 12

default project settings 14, 15

project, creating 13, 14

W

WebStorm

about 17

debugging, in Chrome 23

default files 18-21

index.html file, creating 18

project, creating 18

URL 17

web page, running in Chrome 22

WebStorm file watcher 19

white box tests 157

wildcards, TSD

using 138

Windows Workflow Foundation (WF) 160



Thank you for buying Mastering TypeScript

About Packt Publishing

Packt, pronounced 'packed', published its first book, *Mastering phpMyAdmin for Effective MySQL Management*, in April 2004, and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution-based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern yet unique publishing company that focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website at www.packtpub.com.

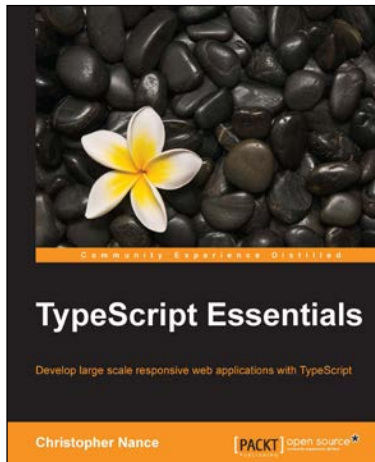
About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around open source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each open source project about whose software a book is sold.

Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to author@packtpub.com. If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, then please contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.

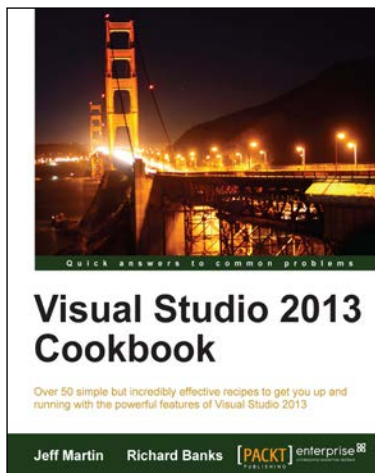


TypeScript Essentials

ISBN: 978-1-78398-576-0 Paperback: 182 pages

Develop large scale responsive web applications with TypeScript

1. Explore the key features of TypeScript to develop web applications of your own.
2. Take advantage of the static typing system to improve the web development experience and add stability to your code.
3. Discover how to effectively use type annotations, declaration files, and ECMA script integration with lots of code and examples.



Visual Studio 2013 Cookbook

ISBN: 978-1-78217-196-6 Paperback: 332 pages

Over 50 simple but incredibly effective recipes to get you up and running with the powerful features of Visual Studio 2013

1. Provides you with coverage of all the new Visual Studio 2013 features regardless of your programming language preference.
2. Recipes describe how to apply Visual Studio to all areas of development: writing, debugging, and application lifecycle maintenance.
3. Straightforward examples of building apps for Windows 8.1.

Please check www.PacktPub.com for information on our titles



Mastering JavaScript Design Patterns

ISBN: 978-1-78398-798-6 Paperback: 290 pages

Discover how to use JavaScript design patterns to create powerful applications with reliable and maintainable code

1. Learn how to use tried and true software design methodologies to enhance your Javascript code.
2. Discover robust JavaScript implementations of classic as well as advanced design patterns.
3. Packed with easy-to-follow examples that can be used to create reusable code and extensible designs.



Bootstrap Site Blueprints

ISBN: 978-1-78216-452-4 Paperback: 304 pages

Design mobile-first responsive websites with Bootstrap 3

1. Learn the inner workings of Bootstrap 3 and create web applications with ease.
2. Quickly customize your designs working directly with Bootstrap's LESS files.
3. Leverage Bootstrap's excellent JavaScript plugins.